Qualitätssicherung von Modelltransformationen - Über das dynamische Testen programmierter Graphersetzungssysteme

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

von
Dipl.-Ing. Martin Simon Wieber
geboren am 3. Mai 1982
in Groß-Gerau

Referent:	Prof. Dr. rer. nat. Andy Schürr
Korreferentin:	Prof. Dr. techn. Gerti Kappel
Tag der Einreichung:	13. April 2015
Tag der mündlichen Prüfung:	17. Juni 2015

D17
Darmstadt 2015

Erklärung laut §9 PromO

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 10. April 2015

Martin Wieber

Kurzfassung

Modelle und *Metamodelle* repräsentieren Kernkonzepte der *modellgetriebenen Softwareentwicklung* (*MDSD*). Programme, die Modelle (unter Bezugnahme auf ihre Metamodelle) manipulieren oder ineinander überführen, werden als *Modelltransformationen* (*MTs*) bezeichnet und bilden ein weiteres Kernkonzept. Für dieses klar umrissene Aufgabenfeld wurden und werden speziell angepasste, domänenspezifische Transformationssprachen entwickelt und eingesetzt.

Aufgrund der Bedeutung von *MTs* für das *MDSD*-Paradigma ist deren Korrektheit essentiell und eine gründliche Qualitätssicherung somit angeraten. Entsprechende Ansätze sind allerdings rar. In der Praxis erweisen sich die vornehmlich erforschten *formalen Verifikationsansätze* häufig als ungeeignet, da sie oft zu komplex oder zu teuer sind. Des Weiteren skalieren sie schlecht in Abhängigkeit zur Größe der betrachteten *MT* oder sind auf Abstraktionen bezogen auf die Details konkreter Implementierungen angewiesen. Demgegenüber haben *testende Verfahren* diese Nachteile nicht. Allerdings lassen sich etablierte Testverfahren für traditionelle Programmiersprachen aufgrund der Andersartigkeit der *MT*-Sprachen nicht oder nur sehr eingeschränkt wiederverwenden. Zudem sind angepasste Testverfahren grundsätzlich wünschenswert, da sie typische Eigenschaften von *MTs* berücksichtigen können. Zurzeit existieren hierzu überwiegend *funktionsbasierte (Black-Box-)Verfahren*.

Das Ziel dieser Arbeit besteht in der Entwicklung eines *strukturbasierten (White-Box-)Testansatzes* für eine spezielle Klasse von Modelltransformationen, den sog. *programmierten Graphtransformationen*. Dafür ist anhand einer konkreten Vertreterin dieser Sprachen ein *strukturelles Überdeckungskonzept* zu entwickeln, um so den Testaufwand begrenzen oder die Güte der Tests bewerten zu können. Auch müssen Aspekte der Anwendbarkeit sowie der Leistungsfähigkeit der resultierenden Kriterien untersucht werden.

Hierzu wird ein auf *Graphmustern* aufbauendes Testüberdeckungskriterium in der Theorie entwickelt und im Kontext des *eMoflon*-Werkzeugs für die dort genutzte *Story-Driven-Modeling*-Sprache (*SDM*) praktisch umgesetzt. Als Basis für eine Wiederverwendung des etablierten Ansatzes der *Mutationsanalyse* zur Leistungsabschätzung des Kriteriums hinsichtlich der Fähigkeiten zur Fehlererkennung werden Mutationen zur synthetischen Einbringung von Fehlern identifiziert und in Form eines Mutationstestrahmenwerks realisiert. Letzteres ermöglicht es, Zusammenhänge zwischen dem Überdeckungskonzept und der *Mutationsadäquatheit* zu untersuchen. Im Rahmen einer umfangreichen *Evaluation* wird anhand zweier nichttrivialer Modelltransformationen die Anwendbarkeit und die Leistungsfähigkeit des Ansatzes in der Praxis untersucht und eine Abgrenzung gegenüber einer quellcodebasierten Testüberdeckung durchgeführt.

Es zeigt sich, dass das entwickelte Überdeckungskonzept praktisch umsetzbar ist und zu einer brauchbaren Überdeckungsmetrik führt. Die Visualisierbarkeit einzelner Überdeckungsanforderungen ist der grafischen Programmierung bei Graphtransformationen besonders nahe, so dass u. a. die Konstruktion sinnvoller Tests erleichtert wird. Die Mutationsanalyse stützt die These, dass die im Hinblick auf Steigerungen der Überdeckungsmaße optimierten Testmengen mehr Fehler erkennen als vor der Optimierung. Vergleiche mit quellcodebasierten Überdeckungskriterien weisen auf die Existenz entsprechender *Korrelationen* hin. Die Experimente belegen, dass die vorgestellte Überdeckung klassischen, codebasierten Kriterien vielfach überlegen ist und sich so insbesondere auch für das Testen von durch einen Interpreter ausgeführte Transformationen anbietet.

Abstract

Models and meta-models represent core concepts of the Model-Driven Software Development (MDSD) paradigm. Programs which modify or translate models with reference to their meta-models are commonly referred to as Model Transformations (MTs). They represent a further vital concept of MDSD and their definitive scope led to the development and usage of dedicated and adapted domain specific transformation languages.

Because of the key role MTs play in the MDSD paradigm their correctness is of utmost importance. Consequently, a thorough quality assurance process is very advisable albeit respective approaches are scarce. Some developed techniques based on formal verification techniques suffer from their inherent complexity and high effort in practice. Also scalability w. r. t. MT size can be problematic which renders some form of abstraction a necessity and also implies neglecting certain details of an implementation. Testing, on the other hand, does not feature these disadvantages. Unfortunately, well-established testing techniques developed for conventional imperative programming languages cannot be (fully) reused due to the different properties of MT languages. Adapted testing techniques considering common MT features are a necessity. Up to now, black-box testing based on input-output-behavior is predominant.

The overall aim of this work is the development of a white-box testing approach for a specific class of MTs called programmed graph transformations. For this purpose a white-box coverage criterion needs to be developed which can then be used to limit the testing effort or to evaluate the quality of a test suite. Applicability and performance aspects of such a criterion should also be evaluated.

For this purpose a new test coverage criterion based on graph patterns is presented. It is mainly described from a theoretical and generic point of view but also practically implemented as part of the eMoflon tool suite targeted at the Story-Driven-Modeling (SDM) language. For reusing the well-established mutation analysis technique as a means for evaluating the performance of the criterion in terms of its error detection capabilities, errors need to be synthesized and implanted via mutation operators newly developed as part of a mutation framework. Consequently, the relation between the coverage criterion and the mutation adequacy can also be examined. Evaluations of the general applicability and the performance are done as part of a larger case study based on two complex transformation scenarios. This also includes a distinction between the new criterion and code based coverage.

Initial experiments show that the new coverage notion is feasible and of practical use. The visual nature of the occurring coverage items is close to the graphical type of programming used in graph transformations which eases the construction of sensible test cases. The results of the mutation analysis back the hypothesis that a test set which achieves a high coverage is likely to discover more errors than a test set with low coverage. Comparisons of the coverage metric values with those of code based coverage suggest the existence of correlations. Measurements also confirm that the new coverage is superior to the considered forms of code coverage in many ways which might prove beneficial in many test setups, especially when testing MTs which are executed by an interpreter.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Liste der Algorithmen

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

(Douglas R. Hofstadter, aus [Hof99, S. 152])

1 Einleitung

Der anhaltende technische Fortschritt führt, aufgrund der damit einhergehenden vermehrten Vernetzung und dem wachsenden Maß an Datenerhebung und -verarbeitung, dazu, dass technische Produkte einerseits immer komplexer werden, sich andererseits aber immer „intelligenter“ an ihre Umwelt und die sich ändernden Anforderungen anpassen sollen. Die rasante Entwicklung der Mikro- und Nanoelektronik der letzten Jahre und Jahrzehnte ermöglicht heutzutage Anwendungen, die noch vor wenigen Jahren von der überwiegenden Mehrzahl der Menschen als pure Science-Fiction abgetan worden wären – man denke nur an diverse mobile Computer, die ein Großteil der Menschheit tagtäglich mit und an sich herumträgt. Selbst in alltäglichen Geräten und Maschinen stecken immer mehr und immer leistungsfähigere Computer in Form *Eingebetteter Systeme*.

Auf all diesen ubiquitären Computern läuft irgendeine Art von (Spezial-)*Software*. Die Bedeutung von Software für den technischen Fortschritt, die Wertschöpfung und die Diversifikation am Markt ist nicht zu unterschätzen, vgl. [BMT11; Wym12] für den Automotive-Sektor, denn Software lässt sich unter anderem im Vergleich zu Hardware flexibel anpassen und ändern. Neue Funktionalitäten, engl. *Features*, lassen sich, in Form von Software-Updates, vergleichsweise schnell und mit geringen Kosten auf eine Vielzahl von Geräten verteilen und so eventuell vorhandene Fehler auch nach einer Auslieferung noch korrigieren. Somit verwundert es nicht, dass, ob der zunehmenden Umsetzung von Lösungen in Software, der Anteil der Entwicklungskosten für diese, bezogen auf die Gesamtkosten der Entwicklung, tendenziell eher ansteigt. Dies ist beispielsweise in der Automobilindustrie zu beobachten, vgl. hierzu die FAST2025-Studie [Wym12, Abb. 24, S. 43 sowie S. 74-78]. Die Autoren der Studie halten allerdings auch einen gegenläufigen Trend aufgrund des Optimierungspotenzials für möglich [Wym12, S. 77].

Der vermehrte Einsatz von Software beinhaltet allerdings auch eine Zunahme der Gefahren aufgrund fehlerhafter Programme. Neben (vornehmlich als peinlich zu bezeichnenden) Image-Schäden durch fehlerhafte Softwareprodukte – man denke beispielsweise an Berichte zu Sicherheitslücken, welche die informationelle Selbstbestimmung von

Anwendern unterhöhlen können – die allerdings auch zum Teil beträchtliche monetäre Schäden bedingen können, sind auch handfestere Risiken für Leib und Leben mit dem vermehrten Einsatz von Software verbunden. Unmittelbar deutlich ersichtlich ist dies im Avionik-Umfeld, den Themenkomplexen *X-by-Wire* oder *Autonomes Fahren* im Automotive-Sektor, im medizinischen und gesundheitsführsorglichen Bereich oder aber bei der Regelung und Steuerung von Großanlagen der Industrie[1] bzw. den – durch den angestrebten Ausbau der Erneuerbaren Energien zunehmend dynamischen Effekten unterliegenden – Stromnetzen.

Im Spannungsfeld zwischen einem wachsenden Bedarf an funktions- und variantenreichen Software-Produkten, den wachsenden Qualitätsanforderungen – auch aufgrund zunehmender regulatorischer und gesetzlicher Auflagen und Bestimmungen – der fachlichen Spezialisierung der Arbeitnehmerschaft, aufgrund des insgesamt anwachsenden relevanten Wissens und angestrebter verkürzter Ausbildungszeiten, sowie dem Innovations- und Kostendruck durch den globalen Wettbewerb, werden *Planbarkeit*, *Kontrollierbarkeit* und *Prozessqualität* immer wichtiger. Das Wissensgebiet der *Softwaretechnik*, engl. *Software-Engineering*, untersucht Techniken und Methoden, um Software-Produkte „ingenieursmäßig“ zu entwickeln. Es werden Verfahren bereitgestellt, die sich in der Praxis als geeignet und angemessen bewährt haben, komplexe Entwicklungsvorhaben erfolgreich abzuschließen. Dabei werden wichtige Teilschritte des Entwicklungsprozesses wie Planung, Entwurf, Realisierung und Produktion, Qualitätssicherung, Auslieferung und Inbetriebnahme sowie Wartung und Stilllegung betrachtet, vgl. z. B. das *V-Modell (XT)* [12; Boe79].

Ein wichtiger Trend in der Softwaretechnik in jüngerer Zeit ist der vermehrte Einsatz von *modellgetriebenen* und *-basierten Verfahren*. So berichten Van Der Straeten et al. in [VMV09] über den erfolgreichen *Model Driven Engineering (MDE)*-Einsatz in der Automobilindustrie bei der Entwicklung von Echtzeit- und Eingebetteten Systemen. Darüber hinaus werden, wie auch in [Moh+09], Herausforderungen für den erfolgreichen Einsatz entsprechender Verfahren in der Industrie betrachtet. Damit lassen sich im Idealfall Produktivitätssteigerungen bei der Entwicklung, eine konstant hohe Qualität der entwickelten Software-Produkte,[2] die Nachvollziehbarkeit und Dokumentation sowie Möglichkeiten zum flexiblen Austausch oder der Unterstützung von Zielplattformen erzielen. Erreicht wird dies – vgl. hierzu z. B. auch [Moh+09; VMV09] – unter anderem durch (i) eine bestmögliche Unterstützung und frühzeitige Einbindung von *Domänenexperten*, welche zum Teil nur über eingeschränkte Erfahrung/Expertise im Bereich der Softwareentwicklung verfügen, in den Entwicklungsprozess, (ii) die Etablierung eines höheren Abstraktionsniveaus mit Hilfe *domänenspezifischer Sprachen*,[3] häufig auf Grundlage einer *deklarativen*, *visuellen*, *Implementierungstechnologie-neutralen* und *formalen* Beschreibung, (iii) die Möglichkeit zur *frühen Verifikation und Validierung* aufgrund von ausführbaren Modellen, (iv) die im Idealfall inhärent anfallende Teildokumentation der ausgelieferten Software-Produkte, z. B. durch Design-Modelle, sowie (v) die *Modellbildung* selbst, welche nur noch die problemrelevanten Aspekte in einer auf das Wesentliche

[1] Diesbezüglich oft genannte Stichworte sind *„Industrie 4.0“* [Bun] und *„Cyber-Physical Systems“* [Bro+10; BCG12].

[2] Konkrete Ausprägungen werden oft als *Artefakte* bezeichnet.

[3] Die Abstraktion erfolgt weg vom Quellcode einer allgemeinen Programmiersprache hin zu einer angepassten Beschreibung, z. B. in einem geeigneten Formalismus (Zustandsautomaten, Petri-Netze – für eine Einführung siehe [RD14] – etc.).

reduzierte Abbildung der Wirklichkeit berücksichtigt und technische Details der konkreten Realisierung ausblendet.

Ein zweiter wesentlicher Aspekt der modellgetriebenen Softwareentwicklung liegt in der vermehrten *automatisierten Erzeugung* von Software-Produkten und der damit einhergehenden Möglichkeit zur Wiederverwendung und dem vermehrten Ausschluss von einfachen Kodierungsfehlern. In [MD08b] identifizieren die Autoren in ihrer Meta-Studie den erhofften Produktivitätszuwachs sowie die Forderung nach gesteigerter Qualität als die beiden Hauptgründe für den Wunsch der Industrie nach einem Einsatz von *Model Driven Software Development (MDSD)*-Verfahren. Das Kernkonzept hierfür bilden Abbildungen zwischen Modellen [SK03], allgemein als *Modelltransformationen* bezeichnet. Dabei unterscheidet man grundsätzlich zwei Arten von Abbildungen: (i) Abbildungen der Art *Modell-zu-Text* (M2T) sind Grundlage vieler generativer Techniken und werden in Form von *Codegeneratoren* (Text aus Modell) und *Parsern* (Modell aus Text) realisiert, (ii) *Modell-zu-Modell* (M2M) beschreiben dagegen Abbildungen zwischen Modellen, beispielsweise von unterschiedlichem Detaillierungsgrad oder verschiedener Sprache. Für eine detailliertere Darstellung dieser Konzepte sei hier bereits auf Kapitel 3 verwiesen.

Der Einfluss modellbasierter Techniken ist nicht nur auf solche Vorhaben beschränkt, in denen unmittelbar auf sie zurückgegriffen wird. Viele kleinere und größere Entwicklungswerkzeuge können modellbasierte Verfahren nutzen oder basieren auf diesen. Dadurch beeinflussen modellbasierte Ansätze unter Umständen indirekt auch Eigenschaften von Artefakten, die selbst nicht modellbasiert entwickelt wurden. Im Hinblick auf eine möglichst hohe Qualität der Endprodukte sollten folglich auch modellbasierte Verfahren aufgrund ihres Einflusses qualitätsgesichert werden. Insbesondere stellen Modelltransformationsbeschreibungen selbst ausführbare Software-Artefakte dar, und sollten im Hinblick auf relevante Qualitätseigenschaften – z. B. im Hinblick auf die *Korrektheit* (im Sinne der Anforderungen) – untersucht werden, vgl. z. B. [AB11]. Im Idealfall sollten sie sogar selbst modellbasiert entwickelt werden [LK11a].

Die Qualitätssicherung modellbasierter Verfahren – insbesondere unter Bezugnahme auf *testende Verfahren* – ist, auch aufgrund des relativ geringen Alters solcher Verfahren, Gegenstand aktueller Forschung. So orientieren sich einschlägige Normen und Standards für oder mit Bezug auf Software und den mit ihr verbundenen Risiken und Qualitätsanforderungen, wie *IEC 61508*,[4] *RTCA DO-178C*,[5] *ISO 26262*,[6] *DIN EN 50128* [7] und *IEC 62304*,[8] was beispielsweise vorgeschriebene oder empfohlene Qualitätssicherungs- und Testverfahren anbelangt, an klassischem Quellcode. Diese Sichtweise greift im Kontext der modellbasierten Entwicklung allerdings oftmals zu kurz, da bereits aus Kostengründen Fehler so früh wie möglich identifiziert werden sollten. In einem mehrstufigen Prozess hin zum ausführbaren Code sollten folglich auch schon modellbasiert entwickelte Zwischenergebnisse so frühzeitig und gründlich wie möglich verifiziert und validiert werden. Die vorliegende Arbeit widmet sich deshalb, wie im nächsten Unterabschnitt detailliert ausgeführt wird, der Qualitätssicherung, genauer, dem *dynamischen Testen* bestimmter Modelltransformationsklassen.

[4] „Functional safety of electrical/electronic/programmable electronic safety-related systems“
[5] „Software Considerations in Airborne Systems and Equipment Certification“
[6] „Road vehicles – Functional safety“
[7] „Bahnanwendungen – Telekommunikationstechnik, Signaltechnik und Datenverarbeitungssysteme – Software für Eisenbahnsteuerungs- und Überwachungssysteme“
[8] „Medical device software – Software life cycle processes“

1.1 Motivation der Arbeit

Modelltransformationen sind in der modellbasierten Softwareentwicklung von zentraler Bedeutung. Mit ihrer Hilfe werden viele der Verheißungen des Paradigmas überhaupt erst möglich. Beispielsweise kann mit ihrer Hilfe eigenen Sprachen/Modellen eine Semantik gegeben werden (z. B. durch Abbildung auf eine Zielsprache mit als bekannt vorausgesetzter Semantik, vgl. z. B. [Che+05a] oder [VMV09, Abschnitt 4.2]). Da Modelltransformationen selbst Programme darstellen, liegt es nahe, bei ihrer Entwicklung ebenfalls modellbasierte Verfahren einzusetzen. Dies ist unter anderem dahingehend gängige Praxis, als dass domänenspezifische Sprachen – hier für die Domäne der Abbildungsbeschreibungen für Abbildungen zwischen Modellen oder zwischen Modellen und Text – zum Einsatz kommen. Typische Vertreter der Tansformationssprachen unterscheiden sich dabei konzeptionell deutlich von klassischen Programmiersprachen (s. Kap. 3 für die Details). Bedenkt man, dass Transformationen einerseits eine zentrale Rolle in der Entwicklung einnehmen, andererseits ihre Beschreibung selbst in Sprachen erfolgt, deren Eigenschaften nur eingeschränkt mit denen klassischer Programmiersprachen vergleichbar sind, lässt sich bereits erahnen, dass *spezifische, angepasste Methoden zur Qualitätssicherung* benötigt werden.

Die beiden Hauptziele der Qualitätssicherung lassen sich mit den erwähnten Begriffen *Verifikation* und *Validierung* (auch *Validation*) umschreiben [Boe84]. Nach [Bal98, S. 101] unterscheiden sich die leider häufig inkonsequent bzw. synonym verwendeten [Som95, S. 446] Begriffe anhand der zugrunde liegenden Problemstellung: Bei der Verifikation wird versucht, die Frage zu beantworten, ob das zu konstruierende Produkt *korrekt* (bezogen auf eine Referenz/Vorgabe, z. B. in Form einer *formalen* Spezifikation) umgesetzt wurde, in dem Sinne, dass es sich *konform* zur Spezifikation verhält. Dabei unterscheidet man die *formale Verifikation*, bei der Eigenschaften – formuliert in einer Sprache mit mathematisch exakt definierter Semantik, vgl. [Som95, S. 159] und beispielsweise gegeben durch Invarianten oder Konformitätsbeziehungen – für eine Implementierung und/oder Spezifikation mittels mathematischer Verfahren zu beweisen sind, von *nichtformalen* Verfahren, wie beispielsweise dem *nicht erschöpfenden Testen*. Im Rahmen der Validierung wird dagegen die Frage nach der *Plausibilität* einer Lösung untersucht [Boe84] also ob die ermittelte Vorstellung des zu konstruierenden Systems (oder das daraus bereits abgeleitete System selbst) tatsächlich zu den praktischen Gegebenheiten, Anforderungen und (Kunden-)Wünschen passt oder, anders ausgedrückt, inwiefern das System überhaupt in der Lage ist, die ihm zugedachte Aufgabe zu erfüllen. Vgl. hierzu auch [Lig09, S. 518].

Im Folgenden werden nur noch Qualitätssicherungsaspekte im Sinne der Verifikation betrachtet. Bezogen auf Modelltransformationen existieren einige Arbeiten, für eine Übersicht s. [Amr+12], die sich mit der formalen Verifikation und dem Nachweis relevanter Eigenschaften (wie z. B. der *Konfluenz* [Küs06; LKC12]) beschäftigen. Grundvoraussetzung hierfür ist die mathematisch exakte Beschreibung, sprich „Formalisierung“ von Transformationen. Ein dazu häufig genutzter Ansatz ist die Theorie der *Graphtransformationen (GTs)* bzw. der regelbasierten *Graphersetzungssysteme*. Formale Verifikationsansätze (nicht nur) für Modelltransformationen haben den Vorteil, unumstößliche Aussagen zu nachweisbaren Eigenschaften zu liefern, zu dem Preis, dass sie in der Praxis nicht immer (praktikabel) anwendbar sind. Oft wird ein gewisses Maß an Wissen, Erfahrung und manuellem Aufwand benötigt, da eine vollständig automatisierte Beweisführung oft nicht möglich ist. Somit erscheinen entsprechende Verfahren in vielerlei Hinsicht teu-

er (können aber unter Umständen trotzdem die *Gesamt*kosten der Entwicklung positiv beeinflussen, vgl. z. B. [Hal90]). Testende Verfahren sind orthogonal dazu zu sehen. In [DPV08] gehen Darabos et al. beispielsweise auf die Vorteile von Testansätzen gegenüber Ansätzen zur formalen Verifikation (jeweils für Graphtransformationen) ein:

> „In general, verification is mainly used in the design phase of transformations, while testing is appropriate in the implementation phase[. . .]. Testing has typically two main advantages: (i) it can be used for large models without combinatorial explosion, (ii) tests are executed directly on the implementation[. . .]“.

In der Praxis haben sich testende Verfahren bewährt und auf breiter Front durchgesetzt, wie sich z. B. anhand der sich hierzu ergebenden Anforderungen aus den einschlägigen Standards mit Bezug zur Softwarequalität (*DO-178C*, *ISO 26262* etc.) belegen lässt, die sie als Stand der Technik auszeichnen. Durch dynamisches, nicht erschöpfendes Testen lassen sich bei klassischen Programmen viele Fehler erkennen, wie durch diverse empirische und experimentelle Untersuchungen nachgewiesen werden konnte, vgl. z. B. [HT90; FW93; Hut+94; Off+96; FI98; FD00; DL00]. Für eine aktuelle, kritische Studie sowie eine ausführliche Zusammenfassung entsprechender Arbeiten jüngeren Datums s. Quelle [IH14]. Das Testen von Modelltransformationen ist dagegen eine noch relativ junge Disziplin [FSB04; LZG05] mit ganz eigenen Herausforderungen [Bau+06; Bau+10]. Insbesondere werden in [Bau+06; Bau+10] folgende Punkte aufgeführt, die eine dedizierte Untersuchung erfordern:

1. Testdatenerzeugung bzw. -generierung
2. Untersuchen und Sicherstellen der Adäquatheit von Tests
3. Orakelfunktionen.

Zu allen drei Aspekten existieren bereits Arbeiten in der Literatur, wobei sich bezogen auf den zweiten Punkt, die gravierendsten Unterschiede zu klassischen Testverfahren ergeben. In [MBT06a, S. 377] stellen Mottu et al. beispielsweise fest:

> „A fundamental step in the elaboration of a test environment for a given software programming paradigm consists of defining criteria to estimate the quality of a test dataset.“

Grundsätzlich besteht eine Möglichkeiten zur Sicherstellung der Adäquatheit von Testmengen (und zur Steuerung der Erzeugung bzw. zur Auswahl geeigneter Tests) darin, eine *Testüberdeckung* zu definieren. Das kann entweder auf Basis von *funktionalen* Anforderungen – den rein ein- und ausgabebasierten *Black-Box*-Kriterien – oder von *strukturellen* Anforderungen – den implementierungsspezifischen *White-* oder *Glass-Box* Kriterien – geschehen. Für das Testen von Modelltransformationen sind beide Optionen zulässig. Allerdings gibt es viele verschiedene Sprachen für Modelltransformationen, wodurch funktionale (Black-Box-)Ansätze besonders attraktiv erscheinen, denen in der Vergangenheit mehr Aufmerksamkeit gewidmet wurde.

Für Graphtransformationen, als spezielle Klasse von Modelltransformationen, sind vergleichsweise nur wenige dedizierte Testansätze in der Literatur beschrieben, vgl. z. B. [DPV08; Gei11; HKM11; KRH12a] oder vgl. Abschnitt 6.3.5. Darüber hinaus existieren auch einige Arbeiten, in denen Graphtransformationsregeln als sog. *Testmodelle*, im

Sinne des *Model-Based Testing (MBT)* [UL07; Dia+07] dazu genutzt werden, um separate Implementierungen zu testen, vgl. z. B. [HL03; HL05; BKS04; GZ05; HM05; HL07; Stü+07a; KRH12b].

Insbesondere zu Beginn des Dissertationsvorhabens, in dessen Verlauf die hier vorliegende Arbeit entstand, war der Aspekt der Adäquatheit von Tests bezogen auf eine unter Qualitätsgesichtspunkten zu untersuchende Graphtransformationsbeschreibung noch wenig erforscht und herausgearbeitet. Aus diesem Grund setzt sich diese Arbeit in wesentlichen Teilen mit Verfahren zur Untersuchung der Adäquatheit von Tests für Modelltransformationen, die mit einer bestimmten Klasse von Graphtransformationssprachen beschrieben sind, auseinander. Hierbei stand im besonderen Maße die Suche nach einem geeigneten *Überdeckungskriterium* für sogenannte *programmierte Graphtransformationen* im Fokus der Betrachtung.

1.2 Zu untersuchende Fragestellungen

Dem Forschungsvorhaben, deren Ergebnisse in dieser Arbeit beschrieben werden, lagen einige Leitfragen zugrunde, die im Rahmen des Forschungsvorhabens zu beantworten waren. Im Folgenden werden diese kurz vorgestellt.

Fragestellung I: *Was soll mit welchem Ziel getestet werden? Welche Eigenschaft ist bzw. welche Eigenschaften sind zu überprüfen und zu zeigen?*

Bei Fragestellung I handelt es sich auf den ersten Blick um eine Frage mit offenkundiger Antwort, etwa dergestalt, dass das dynamische Verhalten einer mittels programmierter Graphtransformationen beschriebenen Transformation zu testen sei! Allerdings können grundsätzlich verschiedenartige Tests durchgeführt und unterschiedliche Systemeigenschaften, sogenannte *Qualitätsmerkmale* [Lig02, S. 5], untersucht werden: in [Tre96, Abschn. 1] werden beispielsweise *Konformität*, *Performanz*, *Robustheit* oder auch *Zuverlässigkeit* als nachzuweisende Eigenschaften aufgeführt; in [Lig02, S. 5 f.] werden darüber hinaus, unter Verweis auf die (mittlerweile veraltete) *ISO/IEC-Norm 9126* [9] sowie auf [Bal98; PS94], die „[...] Qualitätseigenschaften *Sicherheit*, (Zuverlässigkeit), *Verfügbarkeit*, (Robustheit), *Speicher-* und *Laufzeiteffizienz*, *Änderbarkeit*, *Portierbarkeit*, *Prüfbarkeit* und *Benutzbarkeit*“ als nachzuweisende Systemeigenschaften aufgeführt; in [SL05, S. 69, 71] werden Eigenschaften mit Bezug zur Funktionalität – *Angemessenheit*, *Richtigkeit*, *Interoperabilität*, *Ordnungsmäßigkeit*, (Sicherheit) – sowie nichtfunktionale Eigenschaften – (Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit,) *Übertragbarkeit*, *Kundenzufriedenheit* – unterschieden, jeweils ebenfalls unter Bezugnahme auf die *ISO/IEC-Norm 9126*.

Im Rahmen dieser Arbeit liegt der Fokus ganz klar auf dem strukturbasierten Testen mit dem übergeordneten Ziel des Nachweises der Richtigkeit/Korrektheit sowie der Angemessenheit der Implementierung und der Testmenge. Grundsätzlich ist bei Graphtransformationen die sogenannte (Graph-)Mustersuche, engl. Pattern Matching, auf der Basis eines Graphmusters ein wesentlicher Teilschritt. Da hierbei vermehrt mit Fehlern zu rechnen ist, vgl. z. B. [DPV08], sollte vor allem dieser Teilschritt gründlich getestet werden.

[9] Ersetzt durch *ISO/IEC 25000*

Fragestellung II: *Wie können programmierte GTs* systematisch *getestet werden, und zwar unabhängig von eventuell generiertem Quellcode?*

Nachdem durch Fragestellung I die Aufmerksamkeit auf das „Was?“ beim Testen gelenkt wurde, stellt sich mit Fragestellung II nun die Frage nach dem „Wie?“. Gesucht ist ein Verfahren, das auf technische Aspekte eingeht und möglichst automatisiert und unabhängig von subjektiven Entscheidungen eines Testers ist. Für den in Fragestellung II enthaltenen Hinweis darauf, dass nicht auf Grundlage von Quellcode getestet werden soll, existieren zwei Gründe:

1. Die Implementierung zu testender Transformationen erfolgt im zugrunde gelegten Szenario mittels einer Graphtransformationssprache. Somit erscheint es unpassend, auf der Basis daraus abgeleiteter Artefakte, die tendenziell ein niedrigeres Abstraktionsniveau aufweisen, zu testen. Als Analogie für eine Nichtbachtung dieses Zusammenhangs könnte man das Testen eines Programms, geschrieben in einer traditionellen Hochsprache, auf Grundlage des daraus abgeleiteten Assembler-Codes sehen. Dies ist zwar grundsätzlich möglich, erscheint aber wenig intuitiv, zumal die Abbildung zwischen den Sprachen nicht eindeutig ist.
2. Ein generative Ansatz zur Ausführung von Graphtransformationen, bei dem Quellcode entsteht, der die Transformation beschreibt, ist nur eine mögliche Option; alternativ ist eine Ausführung mittels eines *Interpreters* möglich. Dessen Quellcode, falls verfügbar, ist aus Testsicht wiederum völlig unabhängig von der konkreten Transformationsaufgabe zu sehen, wie beispielsweise in [Hil+12] gezeigt werden konnte.

Fragestellung III: *Welche existierenden Ansätze wurden bereits veröffentlicht und wie lassen sich eventuell vorliegende Erkenntnisse und Ansätze wiederverwenden?*

Die Motivation für Fragestellung III erschließt sich unmittelbar. Anstatt „das Rad komplett neu zu erfinden“ (vgl. hierzu [VMV09, S. 41]) sollte die umfangreiche Testliteratur und der Stand der Technik hinsichtlich übertragbarer Methoden untersucht werden.

Fragestellung IV: *Wie lässt sich das Problem konkret lösen, zu entscheiden, wann genug getestet wurde? Wie könnte ein praktikables und effektives Überdeckungsmaß aussehen, so dass die Struktur einer Implementierung eine ausreichend große Berücksichtigung findet?*

In Fragestellung IV spiegeln sich bereits die im Motivationsabschnitt aufgeführten Erkenntnisse wieder. Insbesondere die Frage, ob bzw. wann denn genug getestet wurde, lässt sich mit Hilfe eines geeigneten Überdeckungskriteriums beantworten. Der Möglichkeit zur Abschätzung des bereits Erreichten trägt der Hinweis auf das *Maß* (im Sinne einer *Metrik*) Rechnung. Je näher der Wert der Abdeckungsmetrik an das (theoretische) Maximum heranreicht, desto ausgiebiger und gründlicher gilt das System als getestet.

Warum das Überdeckungsmaß gerade strukturelle Aspekte berücksichtigen soll, wird dann deutlich, wenn man bedenkt, dass strukturbasierte Verfahren eine gute und sinnvolle Ergänzung funktionsbasierter Testverfahren sind. Letztere hatten bereits zu Beginn

des Dissertationsvorhabens eine gewisse Reife erreicht. Darüber hinaus weisen funktionsbasierte Verfahren keinen so unmittelbaren Bezug zu den speziellen Eigenschaften von Modell- und Graphtransformationssprachen auf.

Praktikabel sollte ein Überdeckungsmaß dahingehend sein, dass eine hinreichende Überdeckung – Details hierzu müssen ebenfalls untersucht werden – möglich ist, und dies mit vertretbarem Aufwand praktisch erreicht werden kann. Der Hinweis auf die Effektivität soll andeuten, dass das zu definierende Überdeckungsmaß die Erstellung von Tests fördert bzw. fordert, die in der Lage sind, eventuell vorhandene Fehler mit hoher Wahrscheinlichkeit zu entdecken.

Fragestellung V: *Wie kann eine technische Umsetzung erfolgen? Welche Werkzeugunterstützung ist sinnvoll und möglich?*

Mit Fragestellung V wird der Bezug zu einer Umsetzung des Testansatzes im Rahmen einer existierenden Werkzeuglandschaft des Fachgebiets, an dem diese Arbeit entstand, hergestellt. Testen ist, im Gegensatz zu formalen Verifikationstechniken, in der Anwendung gerade kein ausgesprochen theoretielastiger Ansatz, so dass immer auch praktische Aspekte einer Umsetzung von Interesse sind. Für die Untersuchung der nachfolgenden Fragestellung VI wird, im Falle des Rückgriffs auf die Möglichkeit der praktischen Erprobung, ebenfalls eine Implementierung benötigt.

Fragestellung VI: *Wie verhalten sich entsprechende Überdeckungsmaße und deren Implementierungen in der praktischen Anwendung, also im Rahmen des Testens konkreter Transformationen?*

Sobald ein neues Verfahren entsprechend den Fragestellungen II und IV vorliegt, ist es von unmittelbarem Interesse, wie es sich im Rahmen einer konkreten Benutzung aus Anwendersicht verhält. Werden tatsächlich sinnvolle Testfälle eingefordert? Wie schwer ist es, die Testüberdeckung zu steigern? Wie gut lässt sich die Bestimmung der Testüberdeckung in den Testprozess integrieren? Wie viel zusätzlichen Aufwand bedeutet dies? Welche Werkzeugunterstützung gibt es und welche Unterstützung fehlt eventuell? Wie stabil sind die Implementierungen? All dies sind Teilaspekt der Fragestellung VI.

Fragestellung VII: *Wie kann die Leistungsfähigkeit des zu entwickelnden Testverfahrens (und der beteiligten Überdeckungsmaße) effizient abgeschätzt und überprüft werden, bezogen auf die zu erwartende Adäquatheit der Tests?*

Fragestellung VII zielt auf die Leistungsbewertung des zu entwickelnden Verfahrens ab. Dabei geht es nicht so sehr um vereinzelte konkrete Erfahrungen, sondern eher um eine generelle Abschätzung. Eine Möglichkeit hierzu stellen umfangreiche Benutzerstudien dar, die als schwerwiegenden Nachteil allerdings eine komplexe und aufwendige Durchführbarkeit aufweisen. Der Hinweis auf eine *effiziente* Abschätzung deutet bereits an, dass im Rahmen dieser Arbeit andere Verfahren benötigt werden, die leichter durchzuführen sind. So ist beispielsweise das Einbringen von und die anschließende Suche nach künstlichen Fehlern – mit Methoden des sogenannten *Mutationstestens* [DLS78] erzeugt – ein akzeptiertes Verfahren, um die Leistungsfähigkeit von Qualitätssicherungsansätzen

zu überprüfen [And+06]. Aspekte der Anwendbarkeit eines solchen Ansatzes mussten im Rahmen dieser Arbeit folglich ebenfalls untersucht werden.

Fragestellung VIII: *Wie sehen typische Fehler aus, die in den betrachteten Graphtransformationsklassen auftreten können?*

Bei der Suche eines geeigneten Testansatzes, hilft eine Analyse der zu erwartenden Fehler ggf. dabei, speziell darauf abgestimmte Verfahren zu entwickeln. Dies wäre unter ökonomischen Gesichtspunkten wünschenswert, da so der zu erwartende Nutzen durch die Konzentration auf die wesentlichen Fälle maximiert werden kann. Grundsätzlich setzt auch die Beantwortung von Fragestellung VII mittels der erwähnten mutationsbasierten Verfahren voraus, dass ein gewisses Verständnis für typische Fehlerfälle vorhanden ist, um nicht allzu artifiziell erscheinende Fehler einzubringen. Fragestellung VIII trägt diesen beiden Punkten Rechnung.

Fragestellung IX: *Wie können typische Fehler künstlich erzeugt und deren konkretes Auftreten simuliert werden? Wie lassen sich Transformationsderivate durch Fehlerimplantation*[a] *effektiv und effizient erzeugen?*

[a] Dabei wird das zu testenden System selbst manipuliert, nicht die Daten, auf denen es arbeitet.

Aus Fragestellung VII ergibt sich, dass das Wissen über typische Fehler, welches für die Beantwortung der Fragestellung VIII Voraussetzung ist, noch nicht ausreicht, um experimentelle Nachweise mit aussagekräftigen Ergebnissen zu führen. Um ein Mindestmaß an Aussagekraft sicherstellen zu können, setzen aus Experimenten verallgemeinerbare Erkenntnisse genügend großen Stichproben voraus. Folglich müssen, um nicht auf rein manuelle, aufwendige Verfahren beschränkt zu sein, typische Fehler auch simuliert und automatisiert in eine bestehende Transformationsbeschreibung implantiert werden können. In Fragestellung IX klingt dieser Zusammenhang durch den Hinweis auf die effiziente Erzeugung an, da sich hierzu eine Automatisierung anbietet, vgl. hierzu z. B. [AO08, Abschnitt 8.2].

Fragestellung X: *Lohnt sich ein Einsatz der sich aus Fragestellung II ergebenden Ansätze? Worin besteht der praktische Nutzen und wie signifikant ist er? Wie ist es um die Übertragbarkeit auf ähnliche Problemstellungen bestellt?*

Letztlich führen die Fragestellungen VI bis IX zu der abschließenden Fragestellung X, die sich um Effektivität, Effizienz und Übertragbarkeit dreht. Die Motivation hierfür liegt in einer Bewertung des Erreichten. Der Bezug zur Übertragbarkeit ist dabei deswegen geboten, weil sich alle Ausführungen der Arbeit an einer bestimmten, vorgegebenen Graphtransformationssprache, der sogenannten *Story Driven Modeling (SDM)* Sprache, orientieren; andere (Graphmuster-basierte) Transformationssprachen weisen zum Teil grundlegend andere Eigenschaften auf. Trotz des Bezugs auf die *SDM*-Sprache sollten Konzepte auf ähnliche Sprachen möglichst übertragbar sein, worauf im Verlauf der Arbeit an geeigneter Stelle eingegangen werden wird.

1.3 Wesentliche Ergebnisse

Nachdem der Kontext der Arbeit vorgestellt, die Motivation dargelegt und die wichtigsten zu untersuchenden Fragestellungen genannt wurden, werden nun die wesentlichen Ergebnisse der Arbeit skizziert. Dazu bietet es sich an, von einer idealisierten Darstellung einer archetypischen *Testumgebung* auszugehen, um mit ihrer Hilfe die Beiträge der Arbeit zu verorten. In der Literatur existieren vergleichbare Darstellungen für konkrete Testumgebungen und Transformationssprachen, z. B. [LZG05; DPV08]. In Abbildung 1.1 ist dazu eine generische Testumgebung in Form eines Komponentendiagramms (in UML2.x-Notation, vgl. hierzu auch Kap. 2) skizziert.

An dieser Stelle noch ein kurzer allgemeiner Hinweis zu den verwendeten Bezeichnern innerhalb von Abbildungen im weiteren Verlauf der Arbeit: in Anlehnung an die weit verbreitete Programmierrichtlinie, vorzugsweise englische Bezeichner zu verwenden, werden an einigen Stellen englische Begrifflichkeiten verwendet – vor allem innerhalb von implementierungsnahen Modellen/Diagrammen. Der grundlegenden Verständlichkeit der Arbeit sollten die vereinzelten englischen Begriffe allerdings nicht (wesentlich) schaden.

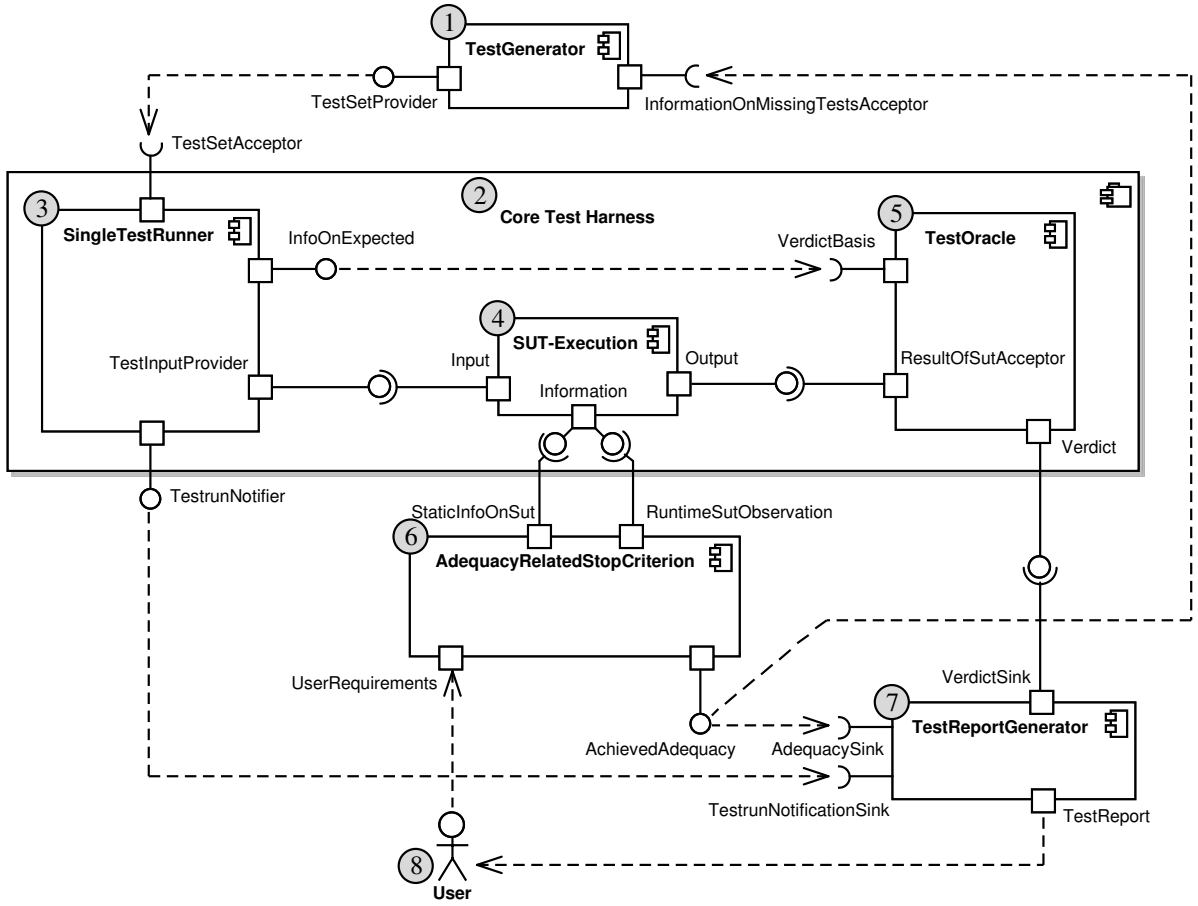


Abbildung 1.1: Übersicht der Komponenten und Verknüpfungen einer vollständigen Testumgebung

Die Testumgebung der Abbildung 1.1 besteht aus folgenden Komponenten: einem Testgenerator ① zur Erzeugung einer Menge von Testfällen, dem Testrahmen ② zur Aus-

führung und Bewertung der Tests, einer Komponente ③ zur Auswahl und Ausführung einzelner Tests der Testmenge, dem eigentlichen *System Under Test (SUT)* ④ (hier also die zu testende Graphtransformation), einem Testorakel ⑤ (hier ist eine Umsetzung mittels Vergleich zwischen tatsächlicher und erwarteter Ausgabe angedeutet), einer Komponente ⑥ zur Messung der Testadäquatheit, durch Beobachtung der Testausführung, und zur Umsetzung eines Stoppkriteriums für das Testen (auf Grundlage von Benutzervorgaben) sowie einer Komponente ⑦ zur Testauswertung und Ergebnisaufbereitung, die Testberichte an den Benutzer ⑧ liefert.

Im Rahmen dieser Arbeit werden folgende Beiträge vorgestellt:

1. Entwicklung eines neuartigen Überdeckungskriterium für das dynamische Testen programmierter Graphtransformationen am Beispiel der Graphtransformationssprache *SDM*; die Überdeckung wird über eine Überdeckung von Graphmustern definiert – bezogen auf die Testumgebung betrifft dieser Beitrag die Komponente ⑥.

2. Implementierung eines Rahmenwerks zur Ableitung, Instrumentierung, Erfassung und Auswertung auf Basis von eMoflon und Eclipse – neben der Komponente ⑥ bezieht sich dieser Beitrag auch noch auf die Komponenten ②, ③ und ⑦.

3. Anpassung und Neuentwicklungen von *SDM*-spezifischen Mutationsoperatoren für das mutationsbasierte Testen – dieser Beitrag *kann* auch zur Umsetzung der Komponente ⑥ benutzt werden; im weiteren Verlauf steht allerdings eine „externe“ (bezogen auf die Abbildung) Verwendung zur Analyse des eigentlichen Überdeckungskriteriums im Fokus der Betrachtung.

4. Implementierung des Mutationsrahmenwerks in Eclipse – dieser Teil ist unabhängig von der eigentlichen, hier abgebildeten Testumgebung zu sehen.

5. Evaluation der Anwendbarkeit anhand einer großen Beispieltransformation mit ersten Erfahrungen auf der Grundlage praktischer Experimente sowie der Leistungsfähigkeit der Überdeckung mittels Mutationsanalyse.

1.4 Aufbau der Arbeit

Die Arbeit gliedert sich im Wesentlichen in drei Teile, die eingerahmt werden von der Einleitung (als Prolog) sowie einem abschließenden Fazit (als Epilog).

In Teil I werden die Grundlagen für diese Arbeit vorgestellt, auf denen die weiteren Untersuchungen und Ergebnisse aufbauen. Dazu gehören eine kurze Einführung in das Themengebiet der *Modellgetriebenen Softwareentwicklung* mit der Einführung der zentralen Begriffe *Modell* und *Metamodell* in Kapitel 2, eine konzeptuelle Vorstellung der zur Beschreibung des dynamischen Systemverhaltens genutzten *Modell-* und *Graphtransformationen* in den Kapiteln 3 und 4, eine übersichtsartige Einführung in den überaus umfangreichen Themenkomplex des Testens von Software in Kapitel 5, unter anderem mit einer Einführung der benötigten Testnomenklatur, sowie einer ausführlichen Auseinandersetzung mit dem aktuellen Stand der Forschung, bezogen auf die Qualitätssicherung von Modelltransformationen inklusive Hinweisen auf und Abgrenzung zu verwandten Bereichen in Kapitel 6.

Teil II der Arbeit ist der Vorstellung der eigentlichen Hauptbeiträge gewidmet, die im Rahmen dieses Dissertationsvorhabens untersucht werden konnten. Analog zu der oben verwendeten Aufzählungsreihenfolge der Beiträge, wird zuerst in Kapitel 7 mit ein auf einer neuen Form der strukturbasierten Überdeckung basierender Testansatz für programmierte Graphtransformationen am Beispiel der *SDM*-Sprache sowie eine zugehörige Implementierung beschrieben. Darauf folgt die Vorstellung des *SDM*-spezifischem Mutationsrahmenwerks in Kapitel 8.

Die Ergebnisse der Untersuchungen zur praktischen Anwendbarkeit der im zweiten Teil vorgestellten Verfahren sowie die Erkenntnisse aus der ersten umfangreicheren Evaluation erfolgt in Teil III. Dazu wurde eine komplexe Transformation getestet, eine existierende Testmenge untersucht und optimiert und abschließend, auf Grundlage einer Mutationsanalyse, eine initiale Abschätzung der Leistungsfähigkeit des Ansatzes vorgenommen. Die dazu durchgeführten Experimente, sowie deren Ergebnisse (inkl. Betrachtung deren Validität) und Implikationen sind in Kapitel 9 beschrieben.

Abschließende Bemerkungen zu dem Erreichten, den daraus ableitbaren Erkenntnissen und deren Übertragbarkeit auf andere Problemstellungen sowie Hinweise auf offen gebliebene Punkte und Fragestellungen, jeweils mit anzunehmender Relevanz für mögliche zukünftige Forschungsvorhaben, erfolgen in Kapitel 10.

Teil I

Grundlagen

Abstraktion ist der Schlüssel, um Komplexität zu verwalten.

(Andrew S. Tanenbaum, aus [Tan09, S. 34])

2 Modellgetriebene Softwareentwicklung

Inhaltlich ist die vorliegende Arbeit im Kontext der modellgetriebenen Softwareentwicklung, kurz *MDSD* [SV05; BCW12], angesiedelt, deren Grundlagen und wichtigsten Konzepte in diesem Kapitel überblicksartig vorgestellt werden. Neben dem *MDSD*-Begriff existieren auch die Bezeichnungen *MDD*, vgl. z. B. [Bal+06; Fra+06], bzw. *MDE*, vgl. z. B. [Ken02; Fav04; Sch06] oder [BCW12, Abb. 2.1, S. 10]. Diese sind etwas breiter gefasst, was sich an dem fehlenden unmittelbaren Bezug zur Softwareentwicklung äußert.

Eine weiterer Begriff mit Bezug zu dem hier betrachteten Themenkomplex ist der der *Model Driven Architecture (MDA)*. Dabei handelt es sich um eine geschützte Wortmarke der *Object Management Group (OMG)*,[1] einer Organisation von Industrieunternehmen und andern interessierten Institutionen, die sich der Entwicklung und Pflege von Standards in den Themenkomplexen *OO* und Modellierung verschrieben haben. Der *MDA*-Ansatz kann als eine spezielle (und proprietäre) Ausprägung des an sich organisations- und firmenneutralen *MDSD*-Begriffs aufgefasst werden, s. [Fav04]. Für weitere Details zu den Begrifflichkeiten sei an dieser Stelle auf die einschlägigen Standards und die Literatur verwiesen [01; 03a; KWB03; SV05; BCW12]).

Das zentrale Konzept des *MDSD*-Paradigmas spiegelt sich in allen erwähnten Bezeichnungen wider: das *Modell*. Modelle sind im Allgemeinen Abstraktionen, also auf das Wesentliche reduzierte Abbilder, insbesondere der Wirklichkeit, vgl. [Lud03]. Als solches bilden sie die Basis für das Verständnis von komplexen Zusammenhängen und Prozessen. Innerhalb der Wissenschaft, aber auch im täglichen Sprachgebrauch, ist der Begriff des Modells vielfach überladen. Auch sonst ist der Modellbegriff schwer zu fassen und eindeutig zu definieren, wie z. B. von Ludewig in [Lud03] festgestellt.

[1] vgl. hierzu `http://www.omg.org`

Selbst innerhalb der Informatik bestehen unterschiedliche Gründe für den Einsatz von Modellen, vgl. hierzu beispielsweise [Lud03] oder auch [Sei03; AK03; Hes06]:

- Ausklammerung irrelevanter Details und Reduktion auf das Wesentliche,
- Entwicklung eines Verständnisses bzw. einer Vorstellung vorher unverstandener oder verdeckter Zusammenhänge durch die Konstruktion eines Modells,
- Dokumentation von Wissen und Erkenntnissen,
- Nutzung als Kommunikationsgrundlage für ein gemeinsames *(Fach-)Vokabular*,
- Verwendung als Grundlage für Konstruktion, Simulation und/oder Tests.

Es ist somit notwendig, den Modellbegriff enger zu fassen. Im *MDSD*-Kontext und im weiteren Verlauf dieser Arbeit sind Modelle *formalisierte*, das heißt, einer bestimmten, stringenten und fest vorgegebenen Form genügende Gebilde bzw. Artefakte. Sie werden in geeigneten *domänenspezifischen* (Modellierungs-)Sprachen mit definierter (abstrakter) Syntax ausgedrückt. Wie Kühne in [Küh06] feststellt, sind solche Modelle damit konzeptuell von *physikalisch-mathematischen Modellen* (z. B. ausgedrückt durch idealisierte oder vereinfachte geometrische Strukturen, Systeme von Differentialgleichungen, Blockschaltbilder der Systemtheorie etc.) und *mathematisch-logischen Modellen* („Interpretation einer Theorie“ im Sinne der Logik) zu unterscheiden.[2] Die zur Modellierung genutzten Sprachen ergeben sich häufig aus der historisch entstandenen und/oder standardisierten Nomenklatur der betrachteten (Problem-)*Domäne*[3] und greifen so Konzepte und Ideen aus dieser unmittelbar auf. Konkrete Modelle beschreiben also einerseits Softwaresysteme und deren Entitäten unter Rückgriff auf Konzepte aus der betrachteten Domäne. Andererseits erfassen Sie häufig auch die Beziehungen der modellierten Systembestandteile untereinander und definieren bzw. beschreiben dadurch deren Bedeutung. Das Voraussetzen einer formalisierten Beschreibung der Modelle ist entscheidend, da dies erst die angestrebte rechnergestützte Verarbeitung ermöglicht. Durch die Verwendung einer formalen Sprache werden auf technischer Ebene (abstrakte) *Syntax* und (statische) *Semantik* der Modelle festgelegt, vgl. [SV05, S. 67]. In Abschnitt 2.2 wird dieser Aspekt noch näher betrachtet werden.

Ein weiterer wichtiger Punkt bei der Nutzung von Modellen, den sowohl Kühne in [Küh06] als auch Ludewig in [Lud03] aufgreifen, ist die Art und Weise, *wie* Modelle genutzt werden. Es existieren zwei Interpretationen: (i) *deskriptive* Modelle *beschreiben* Eigenschaften des Systems, beispielsweise zur „Erfassung von Wissen“, (ii) *präskriptive* Modelle *geben* Eigenschaften des (zu entwickelnden) Systems *vor*, die das System erfüllen muss. Da hier die Erzeugung von korrekter und ausführbarer Software im Fokus liegt, sind die Modelle im weiteren Verlauf überwiegend präskriptiver Natur. Allerdings kommt es auch auf den Nutzungskontext eines Modells an, um entscheiden zu können, um welche Art der Nutzung es sich konkret handelt, vgl. hierzu [Lud03].

Nach diesen bis hier hin eher abstrakten Ausführungen, kommen wir nun zu typischen Vertretern von Modellen und einigen ihrer Anwendungen, die im Rahmen der Softwareentwicklung eingesetzt werden:

- (endliche) Zustandsautomaten zur Beschreibung zustandsbehafteter Systeme,
- Modelle, formuliert in der *Unified Modeling Language (UML)* (vgl. Abschnitt 2.1),

[2] Die Nutzung solcher Modellarten im Rahmen der Software- bzw. Systementwicklung wird damit aber explizit *nicht* ausgeschlossen. Auch *können* die hier betrachteten Modelle *semantisch* solchen Modellen durchaus entsprechen, vgl. dazu ebenfalls [Küh06].

[3] Eine Domäne stellt ein „begrenztes Interessen- oder Wissensgebiet“ [SV05, S. 66] dar.

- *Entity-Relationship*-Modelle [Che76] zur Modellierung relationaler Datenbanken,
- *Message Sequence Charts (MSCs)* zur Beschreibung von Kommunikationsprozessen,
- Petri-Netze [Pet62; Pet76; GV03], z. B. zur Beschreibung dynamischer nebenläufiger (Produktions-)Prozesse,
- Flowcharts bzw. Programmablaufpläne zur Beschreibung von Algorithmen,
- Prozessbeschreibungen anhand der *Business Process Model and Notation (BPMN)* zur Modellierung von Geschäftsprozessen,
- Feature-Modelle zur Modellierung von Variabilität und Konfigurationen,
- MATLAB/Simulink[4]-Modelle zur Beschreibung des Verhaltens ingenieurstechnischer Systeme.

Einer der wichtigen Gründe für den Modelleinsatz im Rahmen der Softwareentwicklung liegt in der effizienten Lösung repetitiver und fehleranfälliger Teilaufgaben und der Reduktion der Beschreibung auf die zu lösenden Aufgabe unter Ausblendung der zugrunde liegende Realisierungstechnologie oder der Zielplattform einer Ausführung. Dabei wird ein höherer Abstraktionsgrad angestrebt, als bei einem Einsatz weit verbreiteter Programmiersprachen (Java, C, C++, usw.), die i. d. R. einen wesentlich größeren Abstand zu der Problemdomäne aufweisen. Im Idealfall ist so die Beschreibung einer Lösung, also der zu entwickelnden Funktionalität, nur noch von den Konzepten, Begrifflichkeiten und Methoden der Problemdomäne geprägt. Die vermeintliche Lücke zwischen der auf Modellebene (deklarativ) formulierten Problemlösung und der Umsetzung als ausführbare Software (z. B. in Form von Quellcode eine *OO*-Sprache) wird durch eine möglichst vollständig automatisierbare (und ggf. mehrstufig verlaufende) *Abbildung* geschlossen. Im *MDA*-Standard werden Abbildungen dieser Art als *Mapping* oder *Transformation* bezeichnet. Wir verwenden im Folgenden den Begriff *MT* für Abbildungen zwischen (strukturierten) Modellen. Strukturierter Text bzw. Code kann in dieser Hinsicht auch als Modell interpretiert werden. Eine dedizierte Einführung von Modelltransformationen folgt im nächsten Kapitel.

2.1 Die Unified Modeling Language - UML

Wenn es um die Modellierung von Software oder softwarelastigen Systemen geht, stößt man eher früher als später auf die *UML* [11b; 11c; Jec+04]. Hierbei handelt es sich um einen Industriestandard der *OMG*, der in der zweiten Hälfte der 1990er Jahre entstand und zurzeit in zwei Sprachversionen (1.x und 2.x) existiert. Zum Zeitpunkt der Erstellung dieser Arbeit findet eine Wechsel der Version 2.4.1 vom August 2011 zur Version 2.5 statt. Erstere Version wurde auch in Gestalt der internationalen Standards *ISO/IEC 19505-1* und *19505-2* normiert. Der Standard beschreibt eine ganze Sprachfamilie mit gut einem Dutzend spezialisierter *Modellierungssprachen*, jede jeweils mit unterschiedlichem Fokus, bezogen auf die Teilaufgaben und -aspekte bei der SW-Entwicklung. Die Teilsprachen sind teilweise älter als die *UML* selbst, da sie nicht alle von Grund auf neu entwickelt wurden. Die Ziele der *UML* fasst das folgende Zitat aus der offiziellen Spezifikation prägnant zusammen:

[4] kommerzielles Produkt der Firma *The MathWorks*.

> „The objective of UML is to provide system architects, software engineers, and software developers with tools for *analysis, design, and implementation of software-based systems* as well as for *modeling business and similar processes*.“ [11b, S. 1]

Der Softwarebegriff ist dabei als sehr umfassend zu verstehen und schließt, neben (ausführbaren) Programmen, beispielsweise auch Test- und Protokollbeschreibungen sowie Dokumentations-, Analyse- und Design-Artefakte mit ein, vgl. [11b; 11c].

Fast alle Sprachen der *UML* besitzen eine grafische konkrete Syntax.[5] Konzepte, die in verschiedenen Teilsprachen genutzt werden können, besitzen eine weitestgehend einheitliche Darstellung bzw. konkrete Syntax. Auf der Ebene der abstrakten Syntax stützen sich alle Sprachen auf einen gemeinsamen Unterbau an wiederverwendbaren Konzepten. Die Beschreibung dieses Unterbaus sowie der eigentlichen Sprachmittel erfolgt im Standard ebenfalls mit Hilfe von Modellen. Genauer gesagt werden *Klassendiagrammen* mit *OCL*-Bedingungen benutzt, welche durch natürlichsprachlichen Text ergänzt sind. Diese Modelle, welche die eigentlichen *UML*-Sprachen definieren, nutzen also selbst eine an *UML* angelehnte Sprache als Modellierungssprache, nämlich die besagten Klassendiagramme. Letztere sind wiederum in einem verwandten *OMG*-Standard, genannt *Meta Object Facility (MOF)*, beschrieben, dem der Abschnitt 2.2.1 gewidmet ist.

Wie in Abbildung 2.1 dargestellt, lassen sich die Modellierungssprachen der UML in zwei Untergruppen aufteilen, nämlich in (a) *Strukturdiagramme* und (b) *Verhaltensdiagramme*. Strukturdiagramme beschreiben *statische* Aspekte und Bestandteile von Systemen und deren (mögliche) Beziehungen untereinander. Sie werden beispielsweise zur Modellierung, Partitionierung oder Beschreibung der Verteilung von *Daten* verwendet. Der wichtigste Diagrammtyp – im weiteren Verlauf dieser Arbeit ebenfalls häufig verwendet – ist das bereits erwähnte *Klassendiagramm*.[6] Klassendiagramme werden genutzt zur Domänenanalyse, dem konzeptionellen Datenentwurf sowie dem objektorientierten Design. Die wichtigsten Notationselemente (*Klassen*, *Attribute*, *Operationen*, *Assoziationen* bzw. *Kompositionen*, *Vererbungsbeziehungen* usw.) sowie die konkrete Syntax des Klassendiagramms werden im weiteren Verlauf anhand von Beispielen eingeführt. Eng verwandt mit den Klassendiagrammen sind die im weiteren Verlauf der Arbeit häufig auftretenden *Objektdiagramme*. Statt der Beschreibung von Typen (im Sinne von Klassen oder Kategorien) und deren Beziehungen untereinander, beschreiben letztere konkrete (endliche) Mengen von Objekten (im Sinne von Entitäten oder Laufzeitobjekten) sowie deren tatsächlich vorhandene Beziehungen. Typischerweise bezieht sich jedes Objekt eines solchen Diagramms auf einen Typen, definiert in einem Klassendiagramm. Der in [Küh06] vorgestellten Unterscheidung nach handelt es sich bei Klassendiagrammen um sog. (schematische) *Typ*-Modelle, bei Objektdiagrammen dagegen um sog. (extensionale) *Token*-Modelle.

Für eine erste Intuition kann man sich Klassendiagramme stark vereinfacht als visuelle Darstellung von Klassen, Methoden, Vererbungsbeziehungen und Referenzen eines Programms einer *OO*-Programmiersprache vorstellen. Objektdiagramme wären dann die Darstellung von Laufzeitinstanzen sowie deren Verknüpfungen durch ein Geflecht von

[5] Eine bekanntere Ausnahme ist die *Object Constraint Language (OCL)* – eine Sprache zur Formulierung von (Rand-)Bedingungen und Invarianten über Objektstrukturen – mit textueller Syntax.

[6] Klassendiagramme ähneln den älteren *ER*-Modellen aus der Datenbankendomäne, sind allerdings komplett unabhängig vom Relationenmodell oder Datenbanken allgemein zu sehen.

gegenseitigen Referenzen als visuelles Diagramm. Allerdings sollte man sich dabei stets bewusst sein, dass dies nur eine eingeschränkte Sichtweise darstellt. *Klassen eines Klassendiagramms* sind nicht immer im Sinne von *Typen bzw. Klassen eines OO-Programms* zu interpretieren. Andere mögliche Interpretationen wären z. B. *Elemente einer Domäne*, *Konzepte einer Modellierungssprache* oder *Tabellen einer Datenbank*,[7] vgl. [Dan02].

Neben den statischen Strukturen ist vor allem die Funktionalität einer Software, also das ihr zugedachte Verhalten, von Interesse. Zur Beschreibung vom Verhalten bietet die *UML* einige Verhaltensdiagramme, jeweils speziell geeignet zur Beschreibung unterschiedlicher Aspekte des Verhaltens mit unterschiedlichem Detailgrad. Die wichtigsten Diagrammarten sind: (i) *Use-Case-Diagramme* zur systematischen Erfassung von Anwendungsfällen und der Funktionalitäten eines zu entwickelnden Softwareprodukts, (ii) *Zustandsautomaten*, engl. State Machines, die sich an Harels *Statecharts* [Har87] orientieren und sich zur Modellierung von zustandsbehafteten, reaktiven Systemen oder Protokollen eignen, (iii) *Aktivitätsdiagramme*, mit denen sich Abläufe, Prozesse und Algorithmen modellieren lassen sowie (iv) *Interaktionsdiagramme*, die allgemein den Austausch von Nachrichten zwischen Entitäten und deren zeitliche Reihenfolge beschreiben können. Interaktionsdiagramme lassen sich wiederum unterteilen, unter anderem in: (a) *Sequenzdiagramme*, die zur Beschreibung zeitlicher Aspekte von Kommunikationsvorgängen genutzt werden und stark den Message-Sequence-Charts [ITU11] ähneln sowie (b) *Kommunikationsdiagramme*, die zur gröberen Übersicht über Entitäten und den Austausch von Nachrichten zwischen diesen genutzt werden. Im weiteren Verlauf der Arbeit werden uns Aktivitätsdiagramme, in abgewandelter Form, noch mehrfach begegnen.

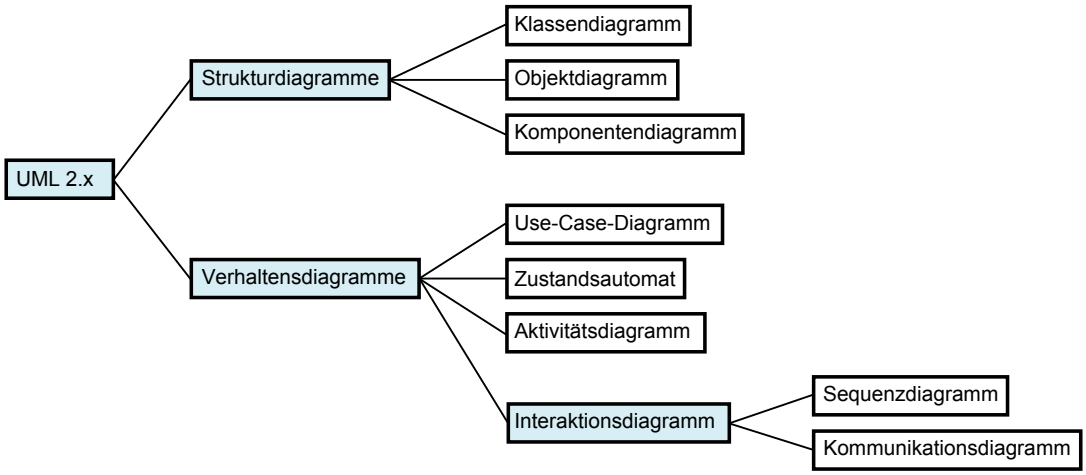


Abbildung 2.1: Die wichtigsten Modellierungssprachen in der UML (Klassifikation nach [11c, Abb. A.5, S. 694])

2.2 Metamodellierung

Eine zentrale Idee der modellbasierten Entwicklung besteht darin, alles durch Modelle zu beschreiben bzw. in Modellen zu erfassen [Béz04]. Im vorangegangenen Abschnitt zur

[7] Im Sinne der erwähnten *ER*-Modelle

UML wurde bereits darauf hingewiesen, dass sich deren Teilsprachen um einen gemeinsamen Kern an Konzepten gruppieren, und dass sich die Gemeinsamkeiten der Teilsprachen in der abstrakten Syntax widerspiegeln. Es wurde auch erwähnt, dass die Modellierungssprachen der *UML* ebenfalls mit Hilfe von Modellen definiert und beschrieben werden. Dahinter steht einerseits die Erkenntnis, dass auch andere Sprachen und Sprachfamilien abseits der *UML* möglich und nicht unüblich sind, und es möglich sein sollte, alle Modellierungssprachen mit einem einheitlichen Beschreibungsstil zu erfassen, vgl. hierzu ebenfalls [Béz04]. Andererseits steht dahinter auch die Idee einer konsequente Anwendung des Modellierungsgedanken „auf sich selbst“.

Die zur Beschreibung von Modellen genutzten „Modelle höherer Stufe“ – solche Modelle also, die selbst eine Modellierungssprache „modellieren“ (im Sinne von beschreiben bzw. definieren) – werden, in Analogie zu *Meta*-Daten, die wiederum (Nutz-)Daten beschreiben, *Metamodelle* genannt [OMG97; 11a; Str98; Sei03; Küh06]. Beispielsweise merkt Seidewitz in [Sei03] an, dass die *UML*-Spezifikation (inklusive der textuellen Beschreibungen) ein Metamodell der *UML*-Sprache(n) darstellt. Favre definiert in [Fav04] ein Metamodell als ein „model of a language of models“, also als ein Modell einer Sprache, in der Modelle formuliert werden.

Es lassen sich dabei, allgemein gesprochen, zwischen den *Elementen* der beteiligten Modelle bzw. zwischen den *Modellen selbst* unterschiedliche Beziehungsarten erkennen und durch mathematische Relationen beschreiben. In [Béz05] beschreibt Bézivin beispielsweise zwei Basisrelationen für den *MDE*-Kontext, nämlich (i) `conformsTo` (zwischen Modellen und den jeweiligen Metamodellen) sowie (ii) `representedBy` (zwischen dem Modellierten und seiner modellhaften Entsprechung). Atkinson und Kühne unterscheiden verschiedene „instanceOf“-Beziehungen [AK03] bzw. Beziehungen der Art „representation“, „instantiation“ und „generalization“ [Küh06]. Später kommen wir auf diese Beziehungskonzepte noch einmal zurück, da sie u. U. dabei helfen können die Idee der Metamodellierung besser zu erfassen.

Atkinson und Kühne stellen in [AK03] auch fest, dass (i) ein Austausch von Modellen über die Grenzen einzelner Werkzeuge hinweg, (ii) die *Abbildungen* bzw. *Transformationen* von Modellen auf andere Modelle sowie (iii) Möglichkeiten zur dynamische Erweiterbarkeit von Modellen, Modellierungskonzepten und Modellierungssprachen Grundvoraussetzungen für eine *Wiederverwendbarkeit* und damit für den Erfolg von *MDD*-Verfahren sind. Sie identifizieren drei Schlüsseltechnologien, die Ihrer Meinung nach notwendig sind, um diese Eigenschaften umzusetzen, nämlich (i) *visuelle* Modellierungssprachen, (ii) einen Rückgriff auf bekannte Konzepte aus der *Objektorientierung* sowie (iii) etwas, das sie „*Metalevel* description techniques“ nennen. Die ersten beiden Punkte wären durch einen Einsatz der *UML* bereits abgedeckt. Vor allem aber der letztgenannte Punkt ist interessant, da er das Konzept der *Metamodellierung* betont.

So, wie verschiedene Realisierungsvarianten durch ein Modell beschrieben werden können, können auch verschiedene Modelle durch ein *Metamodell* beschrieben werden. Dies ist auch von praktischer Bedeutung, da beispielsweise dadurch die Abbildung verschiedener Modelle auf ein einheitliches Serialisierungs- oder Austauschformat erleichtert werden kann. Eine generische Abbildung in ein solches Format kann auf der Grundlage eines gemeinsamen Metamodells beschrieben werden.

Grundsätzlich spricht auch nichts dagegen, dass man den angedeuteten Abstraktionsschritt von Modellen hin zum zugehörigen Metamodell wiederum auf Metamodelle selbst anwendet und so *Meta-Metamodellen* erhält. Eine solche Kette von Abstraktionsschritten

ließe sich beliebig lange fortsetzen, und als Ergebnis erhielte man immer weitere *Ebenen einer Metamodell-Hierarchie* (vgl. [OMG97; AK03] oder auch [11b, Abschn. 7.8 - 7.12]). Für konkrete Metamodell-Hierarchien – insbesondere gilt dies für die konkreten Vertreter der kommenden beiden Unterabschnitte 2.2.1 und 2.2.2 – hat es sich als günstig erwiesen, dass solche Hierarchien durch einen Fixpunkt „nach oben hin“ abgeschlossen sind. Das bedeutet, dass es eine höchste Ebene in der Hierarchie mit festem endlichem Index gibt, und dass sich von diesem Index an Modelle und Metamodelle in Art und Sprache nicht mehr unterscheiden. Modell- und Metamodellsprachen fallen also zusammen.

Um das Konzept der *Metamodell-Hierarchie* zu veranschaulichen, kann man, wie z. B. in [BG01; KBA02] getan, den Vergleich zu den (textuellen) formalen Sprachen ziehen: So stellt beispielsweise eine ausführbare Software eine Konfiguration (Speicherzustand, Register, Programmzähler, Caches, etc.) einer (Hardware-)Rechenmaschine dar. Diese Konfiguration ist im Prinzip eine mögliche konkrete Ausprägung eines abstrakteren Programms oder Algorithmus, welches typischerweise in einer höheren Programmiersprache spezifiziert wurde. Das Programm wiederum kann als konkrete mögliche Ausprägung einer formalen Grammatik interpretiert werden. Letztere kann beispielsweise in der *Extended Backus Naur Form (EBNF)* beschrieben sein.[8] Da die *EBNF* selbst eine formale Sprache ist, kann *EBNF* benutzt werden, um sich selbst zu beschreiben.

2.2.1 MOF

Wie bereits angedeutet, teilen sich die Sprachen der *UML* einen Kern an gemeinsam genutzten Konzepten. Allerdings ist die UML-Spezifikation nicht das einzig mögliche Metamodell (im *OMG*-Umfeld), wie unter anderem Bézivin in [Béz04] bemerkt. Um Metamodelle einheitlich zu erfassen, zu beschreiben und zu dokumentieren, entwickelte die *OMG* den *MOF*-Standard [11a]. In Abbildung 2.2 ist die Metamodell-Hierarchie für *UML* und *MOF* skizziert. Im linken Bereich sind auch die bereits erwähnten Basisrelationen von Bézivin aus [Béz05] dargestellt.

Den Kern des *MOF*-Standards bildet, wie bereits erwähnt, eine reduzierte Form der *UML*-Klassendiagramme als Sprache zur Definition anderer Metamodelle. Ein weiterer Teil des Standards definiert ein *XML*-basiertes Austausch- und Serialisierungsformat, *XML Metadata Interchange (XMI)* genannt [14b], für *MOF*-konforme (Meta-)Modelle. Darüber hinaus existiert eine Standardabbildung von *MOF* auf ein Java-basiertes *Repository* mit standardisiertem *Application Programming Interface (API)*. Letzteres kann *Instanzen* von *MOF*-Modellen aufnehmen und verwalten und bietet per Programmierschnittstelle *Java Metadata Interface (JMI)*[9] [02] die Möglichkeit per Reflexion zuvor unbekannte Metamodelle sowie entsprechende Modelle zu lesen und zu schreiben.

Es gibt zwei Sprachvarianten innerhalb des *MOF*-Standards, nämlich *Essential MOF (EMOF)* und *Complex MOF (CMOF)*. Letzteres basiert auf *EMOF* und komplexeren Konzepten der *UML*, vgl. [11a, S. 47]. Ersteres wurde für eine einfache technische Umsetzung und einfachere Werkzeugunterstützung entwickelt und ist eigenständig nutzbar, vgl. hierzu [11a, S. 31].

[8] Vgl. hierzu den Standard *ISO/IEC 14977:1996(E)*.
[9] Vgl. hierzu den Standard `JSR 040` (entsprechend dem *Java Community Process*).

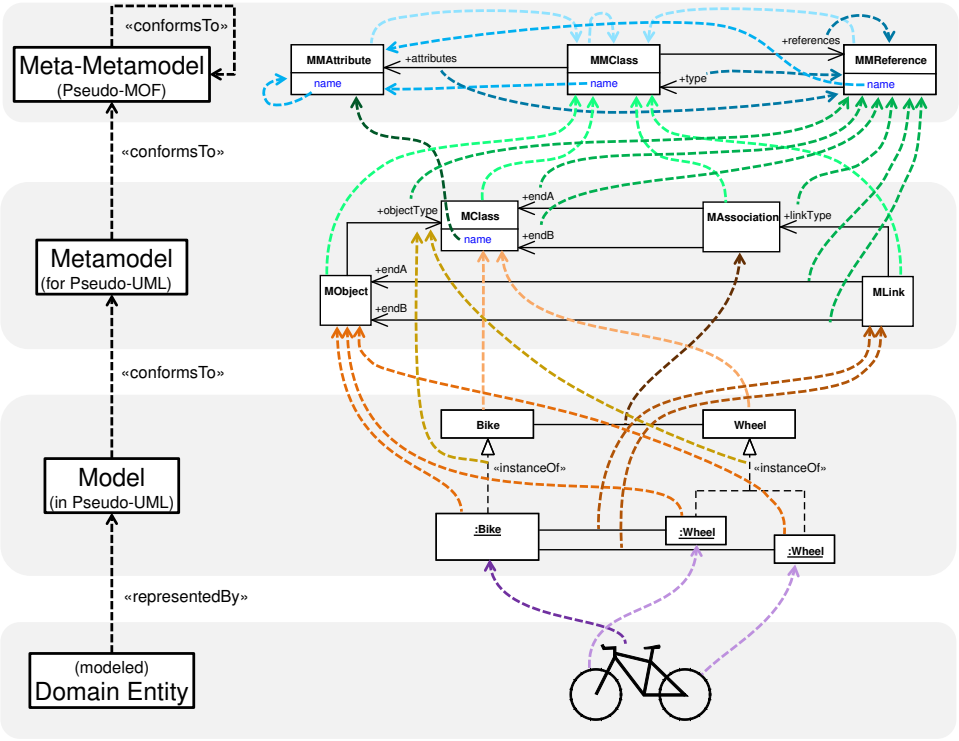


Abbildung 2.2: Metamodell-Hierarchie für UML und MOF nach [11b, S. 20]

2.2.2 EMF

Bei dem *Eclipse Modeling Framework (EMF)*[10] [Ste+09] handelt es sich um ein populäres Rahmenwerk zur Modellierung im Eclipse[11]-Umfeld. Es stellt eine leichtgewichtige und auf unmittelbare Anwendbarkeit hin optimierte Umsetzung des Modellierungsparadigmas zur Entwicklung entsprechender Java-Anwendungen dar. Bei der Entwicklung des Rahmenwerks wurden zentrale Konzepte der *MOF* aufgegriffen, wie beispielsweise der Rückgriff auf Modelle im Sinne einfacher Klassendiagramme:

> „An EMF model is essentially the Class Diagram subset of UML[. . .]“ [Ste+09, S. 16].

Allerdings wurden auch zentrale Aspekte der *MOF* ausgespart, wie beispielsweise die Behandlung von Assoziationen als eigenständige Artefakte mit „eigener Identität“. Dadurch unterscheiden sich auch die Abbildungen von *MOF* mittels *JMI* – beispielsweise auf Basis der Realisierung in MOFLON [Ame+06] – und die Abbildung von *EMF* auf Java zum Teil deutlich voneinander. Der große Erfolg von *EMF* hatte umgekehrt seinerseits auch Einfluss auf die Weiterentwicklung des *MOF*-Standards. *EMOF* ähnelt stark dem *EMF*-Metamodell, welches unter der Bezeichnung *Ecore* bekannt ist, vgl. [Ste+09, S. 30 f.]. Hierdurch lassen sich in *XMI* serialisierte *EMOF*-Instanzen unmittelbar auch im *EMF*-Kontext wiederverwenden.

[10] `http://www.eclipse.org/modeling/emf/` (zuletzt abgerufen am 23.10.2014)

[11] Bei Eclipse handelt es sich um eine ursprünglich von IBM entwickelte, ehemals reine *Integrated Development Environment (IDE)* für die Programmiersprache Java. Im Laufe der Zeit wurde Eclipse weiterentwickelt, hin zu einer Werkzeugplattform und Programmierumgebung für diverse Sprachen.

Neben Ecore bilden vor allem ein generischer Eclipse-basierter Editor (für beliebige Modelle) sowie ein ausgereifter Codegenerator für Java den Kern von *EMF* aus Anwendersicht, vgl. [Ste+09]. Der Editor nutzt zur Visualisierung eine hierarchische Baumstruktur, welche der technisch bedingten Hierarchie der Enthaltenseinsbeziehungen (vgl. hierzu auch die *Composite Aggregations* der *UML* [11c, S. 38]) entspricht, plus eine Anzeige von Objekteigenschaften (in einer *Properties-View*). Der Codegenerator arbeitet auf Basis von *Java Emitter Templates (JETs)* sowie *JMerge* und erzeugt damit aus Ecore-Modellen eine Menge an Java-Schnittstellen und abstrakten sowie konkreten Implementierungsklassen. Im günstigsten Fall wird dadurch bereits der überwiegende Teil des zu realisierenden Programms automatisiert erzeugt. Da allerdings nicht ausgeschlossen werden kann, dass das Ergebnis des Generierungsprozesses im Anschluss manuell angepasst werden muss, wurde dem Codegenerator (mittels JMerge) die Fähigkeit gegeben, Änderungen im Generat zu erkennen und bei einem erneuten Durchlauf beizubehalten, das Generat und den manuellen Anteil also zu mischen (engl. to merge). Daneben besteht in *EMF* laut [Ste+09] auch die Möglichkeit, ein Modell durch Java-*Annotationen* im Quellcode von Interfaces zu definieren. Hierdurch soll der gedankliche Abstand zwischen Programmier- und Modelliertätigkeit gering gehalten werden. Letzterer Gedanke ist ein weiterer Eckpfeiler des *EMF*-Ansatzes, was auch in dem folgenden Zitat seinen Ausdruck findet:

> „With EMF, modeling and programming can be considered the same thing."
> [Ste+09, S. 15]

Auf technischer Ebene biete *EMF* weitere nützliche Funktionen, in Form von (i) Metamodell-basierter Reflexion, (ii) einer mächtigen Laufzeitumgebung mit der Möglichkeit zur automatisierten Registrierung von und Suche nach Metamodellen sowie (iii) eines *Beobachter/Listener*-Konzepts,[12] das auch zur Verfolgung und Nachvollziehbarkeit von Änderungen an Modellen genutzt werden kann. Darüber hinaus gibt es Funktionen zur Überprüfung bzw. Validierung von Modellen, zur Modellpersistierung in Form von *XMI* sowie zur Berechnung von Unterschieden zwischen Modellen.

Der Kern der vorliegenden Arbeit bezieht sich auf *MDSD*-Ansätze im *EMF*-Kontext. Somit werden die zentralen Begriffe *Modell* und *Metamodell* nun in den Definitionen 2.1 und 2.2 entsprechend mit *EMF*-Bezug definiert. Zum besseren Verständnis ist in Abbildung 2.3 die Metamodell-Hierarchie noch einmal für den *EMF*-Fall dargestellt. Im Unterschied zu Abbildung 2.2 kommt auf Modellebene keine Sprache der *UML* zum Einsatz. Das Metamodell ist hier frei selbst definiert und als verschieden von der *UML*-Spezifikation anzusehen. Statt *MOF* wird *Ecore* als Meta-Metamodell genutzt.

Die Hierarchie entspricht im Wesentlichen den oberen drei Stufen von dem, was Bézivin in [Béz04] „3+1 Architektur" nennt. Im linken Bereich der Abbildung wird die konkrete Syntax genutzt, im rechten Teil die abstrakte Syntax. Auf der Modellebene sind die sich entsprechenden Teile farblich hervorgehoben. Zwischen den Elementen des Metamodells und des Meta-Metamodells (in konkreter Syntax) ist durch die Bepfeilung hervorgehoben, wie Metamodellelemente durch Konzepte des Meta-Metamodells modelliert sind.

[12] In *EMF* werden die *Beobachter* aufgrund einer ihrer spezielleren Nutzungsarten als *Adapter* bezeichnet.

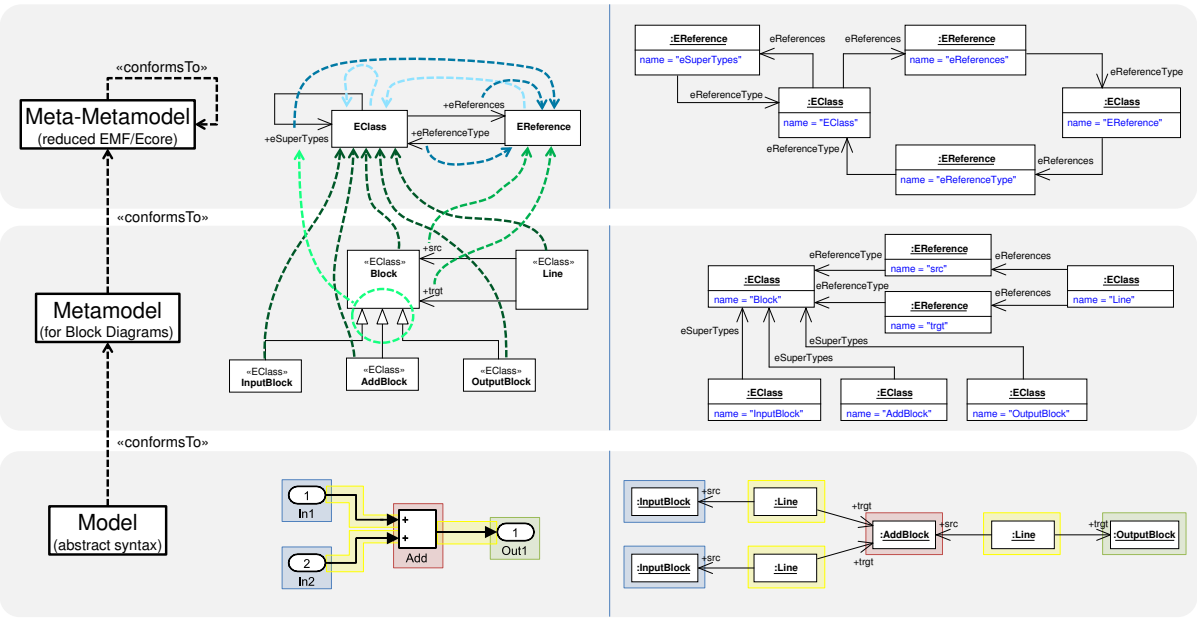


Abbildung 2.3: Metamodell-Hierarchie für EMF am Beispiel eines Blockschaltbilds

Definition 2.1 (Modell, angelehnt an [BET08, Def. 1]):

Ein Modell *ist im Folgenden als „Instanz eines EMF-Modells“ (nach EMF-Sprechweise) zu verstehen, und kann als Objektdiagrammrepräsentation der abstrakten Syntax eines modellierten Artefakts aufgefasst werden.*

Ein Modell ist konsistent *zu seinem Metamodell; letzteres wird auch als* EMF-Modell *bezeichnet.*

Ein Modell ist genau dann als konsistent zu seinem Metamodell anzusehen, wenn die folgenden Bedingungen eingehalten werden:

1. Jedes Objekt referenziert genau eine konkrete Klasse (aus dem Metamodell) als seinen Typ.
2. Jedes Objekt umfasst eine vollständige Belegung aller der durch seinen Typ vorgegebenen Attribute mit zulässigen Werten.
3. Zu jedem Link des Modells gibt es eine passende Assoziation[13] im Metamodell, zu der der Link konsistent ist. Dies impliziert, dass der Link nur dann existieren darf, wenn er für einen gültigen Eintrag in der hypothetischen Relation steht, welche durch die zugehörige Assoziation über den Instanzen der durch sie verknüpften Klassen etabliert wird. D. h., es können nur solche Objekte $o_1, o_2 \in V_O$ durch einen Link $l \in E_L | \mathfrak{a}(l) = a$ überhaupt verbunden sein, deren Typen $t_1 = \mathfrak{t}(o_1), t_2 = \mathfrak{t}(o_2)$ direkt oder aufgrund von Vererbung durch die Assoziation a im Metamodell verbunden sind *und* bei denen die Enden des Links entsprechend der Assoziationsenden festgelegt sind:

$$((\mathfrak{t} \circ s(l) \in clan \circ end_A \circ \mathfrak{a}(l)) \wedge (\mathfrak{t} \circ t(l) \in clan \circ end_B \circ \mathfrak{a}(l))) \vee$$
$$((\mathfrak{t} \circ s(l) \in clan \circ end_B \circ \mathfrak{a}(l)) \wedge (\mathfrak{t} \circ t(l) \in clan \circ end_A \circ \mathfrak{a}(l)))$$

[13] Technisch gesehen, handelt es sich dabei um einzelne oder Paare von zwei verschränkten Referenzen.

vgl. [BET08, Def. 3] (oder auch [AS08, Constraint 1]). Dabei ist $clan(.)$ der Vererbungsclan, also die Menge aller Unterklassen plus die Klasse selbst. Und $end_{A|B}(.)$ bezeichnet die Klasse an einem der beiden Enden einer Assoziation (hier mit den Indices A, B identifiziert).

4. Zusätzliche Einschränkungen aufgrund von Multiplizitäten an den Assoziationsenden sowie durch die speziellen EMF-Containment-Kanten, welche die Anzahl der angrenzenden Links für einzelne Objekte limitieren, werden eingehalten (Zyklen solcher Kanten sind ausgeschlossen und ein Objekt kann nur maximal Bestandteil eines anderen Objektes sein).

Definition 2.2 (Metamodell):

Ein Metamodell[a] *ist hier eine Instanz des* EMF/Ecore-Modells *(vgl. Abbildung D.1). Es kann als Klassendiagramm-hafte Definition der zulässigen Typen und Assoziationen von Instanzmodellen* und *als zum Ecore Meta-Metamodell konsistentes Objektdiagramm aufgefasst werden.*

[a] in der Literatur auch als *Typgraph* [Ehr+06] oder *Graph-Schema*, vgl. z. B. [Rad00], bezeichnet

Der in Definition 2.2 zugrunde gelegte Konsistenzbegriff zwischen Metamodell und Ecore ist dabei der Gleiche, wie zwischen Modell und Metamodell.

2.2.3 Metamodell und Modell am konkreten Beispiel

In Abbildung 2.4 ist ein vollständiges Beispiel-Metamodell gegeben, welches einfache Blockdiagramme beschreibt, wie sie beispielsweise in Matlab/Simulink vorkommen. Das Metamodell bildet den Kern eines umfassenden Beispiels, das im weiteren Verlauf der Arbeit als Anschauungsobjekt für die Vorstellung der im Rahmen dieser Dissertation entwickelten Methodik dient. Es umfasst konkret die wesentlichsten Konzepte und Beziehungen für Blockdiagramme, mit denen einfache mathematische Funktionen auf Basis der arithmetischen Grundoperationen (Summe und Multiplikation) modelliert werden können.

Die Klasse `BDFile` repräsentiert eine Datei des Dateisystems, welche den Inhalt eines konkreten Blockdiagramms speichert, und dient als Wurzel des Containment-Baums. Sie ist Container für eine Menge von `RootSystem`-Instanzen. Letzterer ist eine der beiden konkreten Ausprägungen der abstrakten `BDSystem`-Klasse. *Systeme* beschreiben Funktionseinheiten in Blockdiagrammen, und können Matroschka-artig ineinander verschachtelt werden. Sie dienen dabei der Kapselung von (Teil-)Funktionalitäten. Hier beschreibt ein System genau eine mathematische (Teil-)Funktion. Letztere kann wiederum Teil einer umfassenderen Funktion sein. Falls ein `BDSystem` Bestandteil einer übergeordneten `BDSystem`-Instanz ist, so handelt es sich konkret um ein `SubSystem`. Das „oberste“, äußerste `BDSystem` ist das bereits erwähnte `RootSystem`.

Ein `BDSystem` umfasst *Blöcke* und *Verbindungen* zwischen diesen, entsprechend dem gerichteten Fluss von Daten zwischen Blöcken. Blöcke werden durch die abstrakte Klasse `Block` sowie deren Unterklassen im Metamodell repräsentiert. Verbindungen werden durch Instanzen der Klasse `Line` modelliert. Dabei starten und enden Verbindungen nicht direkt an Blöcken, sondern an zugehörigen Konnektoren, also an logischen Anknüpfungspunkten. Diese sind durch die abstrakte Klasse `Port` sowie durch die beiden konkreten

Spezialisierungen `Outport` und `Inport` modelliert. Eine `Line`-Ausprägung verbindet genau einen `Outport` (Assoziationsende `srcProt`) mit einer Menge von `Inport`-Instanzen (Assoziationsende `trgtPort`). An einem `Outport` können dabei durchaus mehrere Verbindungen entspringen.

Das Metamodell enthält mehrere direkte und abstrakte Unterklassen von `Block`, nämlich (i) `SourceBlock`, welcher einzelne Eingabewerte für die Auswertung der Funktionalität repräsentiert, mit den konkreten Unterklassen `In` und `Constant`, (ii) `SinkBlock`, welcher Datensenken modelliert, und mit `Out` genau eine konkrete Unterklasse besitzt, und (iii) `Operator`, welcher einzelne Verknüpfungsoperationen modelliert, die jeweils Eingangsdaten entgegennehmen, um daraus das Ergebnis für die Ausgangsseite abzuleiten. Konkrete `Operator`-Instanzen können hier vom Typ `Add` (Addition), `Mult` (Multiplikation) oder `Gain` (Multiplikation mit einer Konstanten) sein. Darüber hinaus gibt es im Metamodell mit den Klassen `SystemRef` und `SubSystemBlock` noch zwei weitere konkrete Unterklassen von `Block`, die dazu gedacht sind, Systemdefinitionen an verschiedenen Stellen im Modell wiederverwenden zu können. Mit Hilfe eines `SystemRef`-Blocks lässt sich eine `RootSystem`-Instanz referenzieren und so deren Funktionalität aufrufen. Hierzu besitzt die Klasse `SystemRef` eine unidirektionale Referenz `rootSystem` auf die Klasse `RootSystem` mit der Multiplizität 1. Ein `SubSystemBlock` dagegen beinhaltet ein `SubSystem`, welches wiederum Blöcke – also auch weitere `SubSystemBlock`-Instanzen – enthalten kann. Außerdem werden die Inports und Outports der `SubSystemBlock`-Instanz mit entsprechenden `In`-Instanzen und `Out`-Instanzen über Links zu den entsprechende Assoziationen zwischen `Inport` und `In` respektive `Outport` und `Out` verknüpft, vgl. die Referenzpaare `dataTo` und `inDataFrom` bzw. `dataFrom` und `outDataTo`.

In Abbildung 2.5 ist zusätzlich ein konkretes, konsistentes Modell dargestellt. Im oberen Teil der Abbildung (vgl. 2.5a) in *konkreter* (Simulink-)*Syntax*. Es umfasst einen Eingabeblock (*In1*), einen Subsystemblock (*Subsystem*), und einen Ausgabeblock (*Out1*). Das Subsystem enthält ebenfalls jeweils einen Ein- und einen Ausgabeblock. Darüber hinaus umfasst es einen Verstärkerblock (Verstärkungsfaktor 10). Im unteren Teil der Abbildung (vgl. 2.5b) ist das Modell in *abstrakter Syntax* als Objektgeflecht dargestellt (die Art der Darstellung orientiert sich an der *UML*-Objektdiagramm-Notation). Der für *EMF*-typische Containment-Baum wurde visuell besonders hervorgehoben.

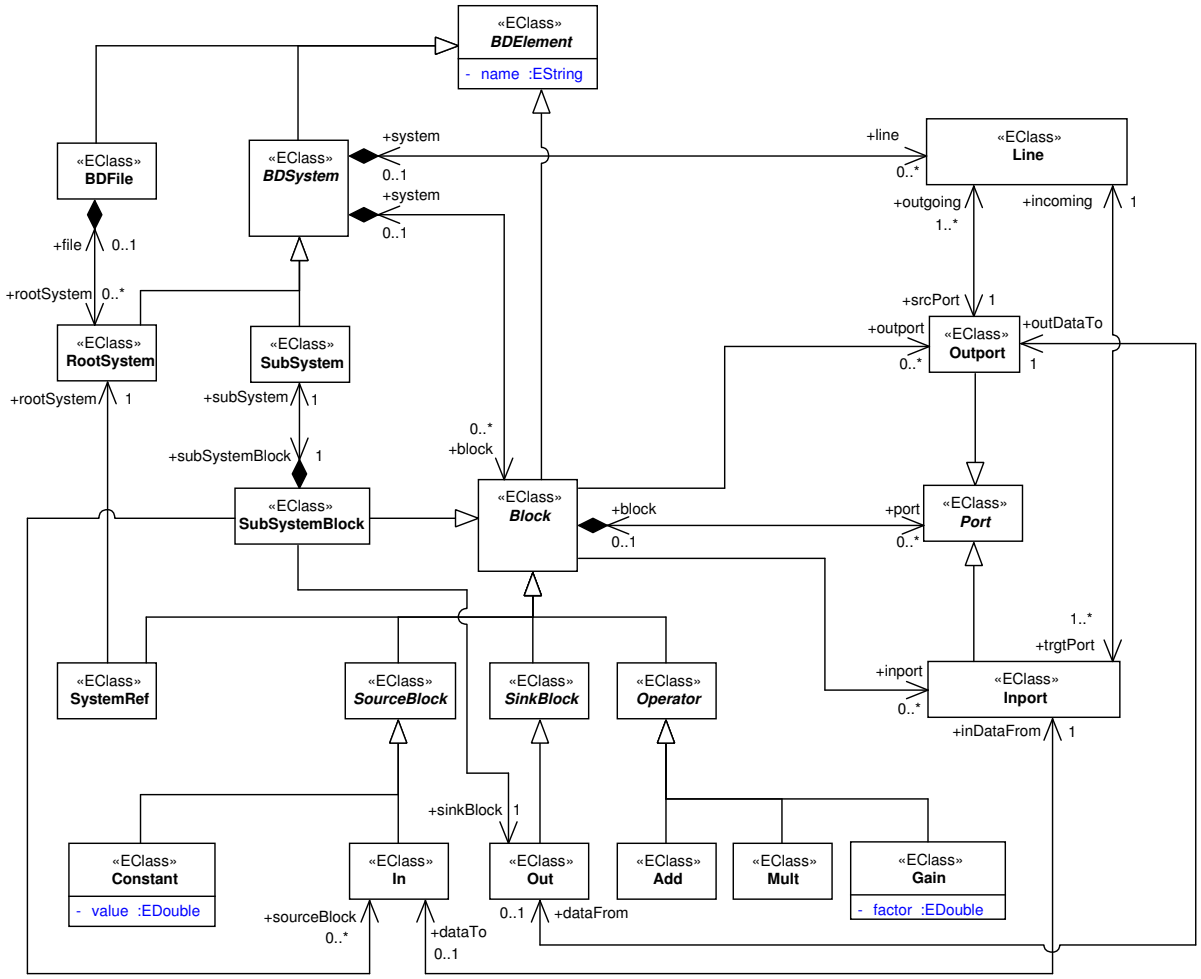
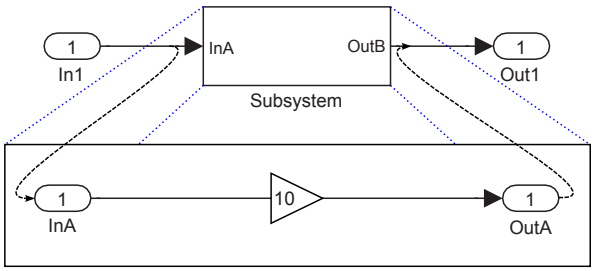
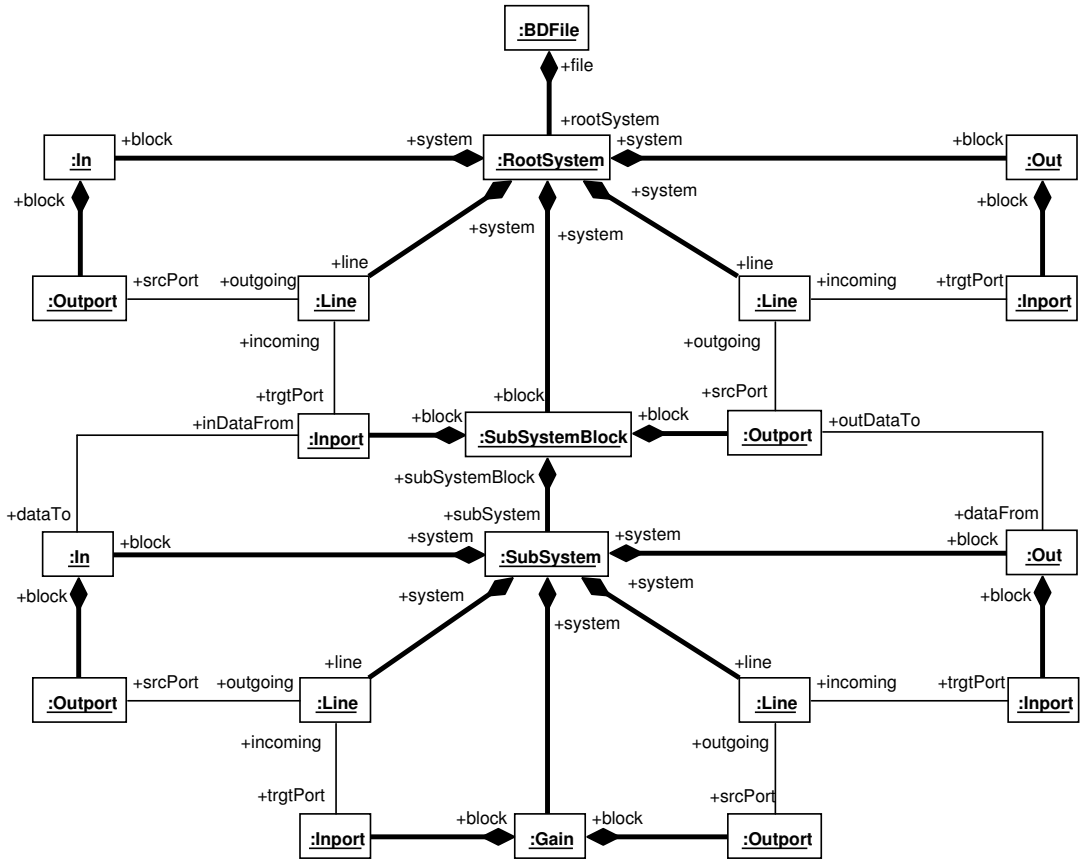


Abbildung 2.4: Metamodell für einfache Blockdiagramme



(a) Einfaches Blockdiagramm in konkreter (Simulink) Syntax



(b) Blockdiagramm in abstrakter Syntax (Objekte und Verbindungen)

Abbildung 2.5: Ein einfaches Blockdiagramm zum Metamodell aus Abb. 2.4

> *Model transformation* is the *process* of converting one model to another model [...]
>
> *(aus dem MDA-Guide der OMG [03a])*

3 Modelltransformationen

Die im vorherigen Kapitel eingeführten Modelle und Metamodelle dienen der Beschreibung von Strukturen bzw. Sprachen. Somit werden durch sie vornehmlich *statische* Aspekte und Eigenschaften von Softwaresystemen beschrieben. Ebenfalls zu diesem Themenkomplex gehört der Begriff der *statischen Semantik* für statisch überprüfbare Konsistenzbedingungen, der ursprünglich im Kontext der Analyse textueller Sprachen geprägt wurde, vgl. z. B. [WM97, S. 228], und der ebenfalls im Zusammenhang mit Metamodellen inkl. zusätzlicher *OCL*-Bedingungen angeführt wird, z. B. in [SV05, S. 68].

Um mit Modellen konkrete Aufgaben zu lösen, müssen erstere übersetzt, verfeinert, komponiert und verarbeitet werden können. Ferner existieren, wie bereits dargelegt, ausführbare Modellarten mit einer *dynamischen Semantik*. Diese beiden Zusammenhänge – das Manipulieren von außen und innere Zustandübergänge – gehen einher mit einer *dynamischen*, schrittweisen Veränderung von Modellen über der Zeit.

Zur Beschreibung solch dynamischer Aspekte existieren unterschiedliche technische Lösungen; auf eine bestimmte Klasse entsprechender Lösungen soll in diesem und im nächsten Kapitel genauer eingegangen werden. Auch existiert mit der QVT-Spezifikation [OMG11] der *OMG* ein Standard aus dem *MDA*-Komplex hierzu. Vor der Betrachtung spezifischer Details erfolgt hier allerdings zuvor noch eine grundlegende Charakterisierung von Ansätzen zur Beschreibung der „Modelldynamik“.

Gemeinsam ist den Dynamikbeschreibungen, dass sie grundsätzlich als *Transformationen* von Modellen – im Sinne von Umformung, Umwandlung, Umgestaltung (s. Duden [04]) – aufgefasst werden können. Transformationen sind, neben (Meta-)Modellen, ein zweiter, wesentlicher Grundpfeiler des *MDSD*-Ansatzes.[1] Sie erst ermöglichen den im letzten Kapitel erwähnten hohen Grad an Automatisierung – sind also *die* Grundvoraussetzung zur Erfüllung der *MDSD*-Verheißungen.

[1] Im *MDA*-Kontext werden beispielsweise Abbildungen von *Platform Independent Model (PIM)* zu *Platform Specific Model (PSM)* sowie von *PSM* zu Code durch Transformationen beschrieben (vgl. z. B. [KWB03]).

Im Folgenden bezeichnet der Begriff *Modelltransformation (MT)* sowohl die Beschreibung bzw. Implementierung als auch den eigentlichen Prozess der automatisierten Ausführung einer Modell*abbildung* (engl. *Mapping*[2]) [03a; SK03]; in Definition 3.1 wird der Begriff entsprechend eingeführt. Welche Bedeutung konkret vorliegt, ergibt sich aus dem jeweiligen Zusammenhang oder ist, falls unklar, explizit angegeben.

Definition 3.1 (Modelltransformation, angelehnt an [03a; SK03]):

Eine Modelltransformation *ist*

(i) *ein* Programm *(bzw. eine Beschreibung hiervon), dessen Eingabe aus einer (i. d. R. einelementigen) Menge von Modellen besteht, welche durch dieses automatisch verarbeitet werden, und dessen Ausgabe entweder aus einer Menge von Modellen (Modell-zu-Modell, M2M) oder textuellen Artefakten (Modell-zu-Text, M2T) besteht oder*

(ii) *der* Prozess der Ausführung *eines solchen Programms mit konkreten Daten.*

Technisch gesehen, handelt es sich bei *MTs* also um eine spezielle Klasse von Programmen, deren wichtigstes Merkmal darin besteht, dass sie Modelle (insbesondere im Sinne von Definition 2.1) modifizieren, übersetzen oder neu erzeugen. Falls die Modelle ausführbar sind oder Teile von ausführbaren Programmen beschreiben, stellen *MTs* nach Aussage von Czarnecki und Helsen eine Form der „*Metaprogrammierung*“ dar [CH06]. Gemeint sind damit Programme, die wiederum selbst ausführbare Programme bzw. wesentliche Bestandteile manipulieren. Folglich sind *MTs* verwandt mit *Programmtransformationen* (vgl. [WM97, S.535]) zur Übersetzung oder Transformation von in verbreiteten Programmiersprachen verfassten Programmen.

Eine zentrale Eigenschaft von nützlichen Transformationen liegt in der ausreichenden Berücksichtigung der Modell*semantiken*. Insbesondere sollte gegeben sein, dass durch eine *MT* entweder (i) eine der Eingabe innewohnende Semantik bei der Transformation zur Ausgabe hin erhalten bleibt, vgl. hierzu [KWB03, S. 24] respektive den auf das Ein- und Ausgabeverhalten bezogenen Äquivalenzbegriff aus [WM97, S. 479], oder aber (ii) eine nützliche Modellsemantik etabliert wird (durch Abbildung auf eine Modellsprache mit bekannter Semantik oder durch die Beschreibung der Übergangsfunktion einer Transitionssystems). Dieser Aspekt ist deshalb von besonderem Interesse, da (a) die Berücksichtigung der Semantik der beteiligten Modelle (welche von Fall zu Fall unterschiedlich sein kann) eine potentielle Fehlerquelle bei Transformationen darstellt und da (b) ggf. mittels der Überprüfung semantischer Äquivalenzen (z. B. von Ein- und Ausgabe) die *Korrektheit* der *MT*-Ausgabe bewertet werden kann.

3.1 Eigenschaften von Transformationssprachen

Zurzeit werden viele spezielle *MT*-Sprachen und entsprechende Werkzeuge gepflegt und entwickelt bzw. weiterentwickelt, vgl. hierzu entsprechende Aussagen in [CH06] bzw. den aktuell und in jüngerer Vergangenheit regelmäßig stattfindenden *Transformation Tool Contest (TTC)*.[3] Ein allgemein akzeptierter (De-facto-)Standard für eine *MT*-Sprache

[2] Dieser Begriff wird unter anderem im *MDA*-Kontext verwendet (vgl. [Ger+02; KWB03]).

[3] `http://www.transformation-tool-contest.eu/` (zuletzt abgerufen am 05.04.2015)

hat sich noch nicht herauskristallisiert, wie beispielsweise in [REP12] festgestellt.

Ganz allgemein lassen sich Modelltransformationen und die dazu verwendeten Transformationssprachen und Werkzeuge anhand ihrer Eigenschaften klassifizieren, wie in [CH03; MCV05; CH06; MV06] durch die Vorstellung entsprechender Klassifikationen gezeigt. Siehe auch [TC10] für einen eingehenden Vergleich. In Anlehnung an diese Klassifikationen sind, je nach Problemstellung, die folgenden Eigenschaften einer *Transformationssprache* sowie deren technischer Basis relevant (z. B. für die Auswahl einer Sprache):

- Paradigma der Sprache (deklarativ, operational, hybrid),
- Ausdrucksmächtigkeit (Turing-vollständig oder eingeschränkt),
- Verfügbarkeit einer (eingängigen) Formalisierung,
- Technologische Basis und Ausgereiftheit,
- Textuelle oder visuelle konkrete Syntax,
- Unterstützung von Bidirektionalität und/oder Inkrementalität,
- Unterstützung von *Higher-Order Transformations (HOTs)* – also Transformationen von Transformationsmodellen,
- Wiederverwendbarkeit durch Vererbungskonzepte und Komponierbarkeit durch ein Modularisierungskonzept,
- Erweiterbarkeit der Sprache,
- Standardisierung und Verfügbarkeit, etc.

Für eine Übersicht von *MT*-Sprachen und deren Klassifikation sei auf die oben aufgeführten Quellen verwiesen.

3.2 Eigenschaften der Transformationsaufgabe

Die modellbezogenen Tätigkeiten, die mit Hilfe von *MTs* gelöst werden können, sind vielfältig. Lano und Clark identifizieren in [LC08] beispielsweise sechs wiederkehrende Szenarien: (i) *Refinements* – Abbildungen allgemeinerer hin zu spezielleren Modellen oder Einschränkungen durch die Einführung restriktiverer (Neben-)Bedingungen, (ii) *Quality Improvements* – Verbesserung der Modellstruktur bzw. -aufteilung, (iii) *Elaboration* – ausschließliche Erweiterungen von Modellen durch Hinzufügen, ohne dass sich dadurch Verschiebungen in der Abstraktionsebene oder Einschränkungen bzgl. der Gültigkeit von ursprünglich möglichen Modellen ergeben, (iv) *Re-expressions* – Übersetzung eines Modells in ein Modell einer anderen Sprache,[4] (v) *Abstractions* – Entfernen bzw. Ausblenden nicht benötigter (technischer) Details, (vi) *Design Patterns* – ähnlich zu Quality Improvements, aber beschränkt auf Umbauten zum Einführen von bekannten, allgemein akzeptierten Teillösungen.

Transformationsaufgaben lassen sich darüber hinaus anhand folgender Kategorien klassifizieren, s. insbesondere [MCV05; MV06]:

- Anzahl und Sprache(n) der Ein- und Ausgangsartefakte,
- Endogen (rephrasing[5]) gegenüber exogen (translation[5]) Abbildungen,
- Modifizierende In-place- oder neu generierende Out-place-Transformationen
- Horizontale (gleichbleibendes Abstraktionsniveau) oder vertikale (unterschiedliches Abstraktionsniveau der Eingabe(n) und Ausgabe(n)) Abbildungen,

[4] Diese Art der Abbildung kann aufgrund semantischer Unterschiede ggf. nur unvollkommen sein.
[5] Diese Begriffe stammen aus dem Gebiet der *Programm*transformationen, s. [Vis01].

- Überbrücken *technologisch* verschiedenartiger Domänen,[6]
- Unidirektionalität gegenüber Bidirektionalität.

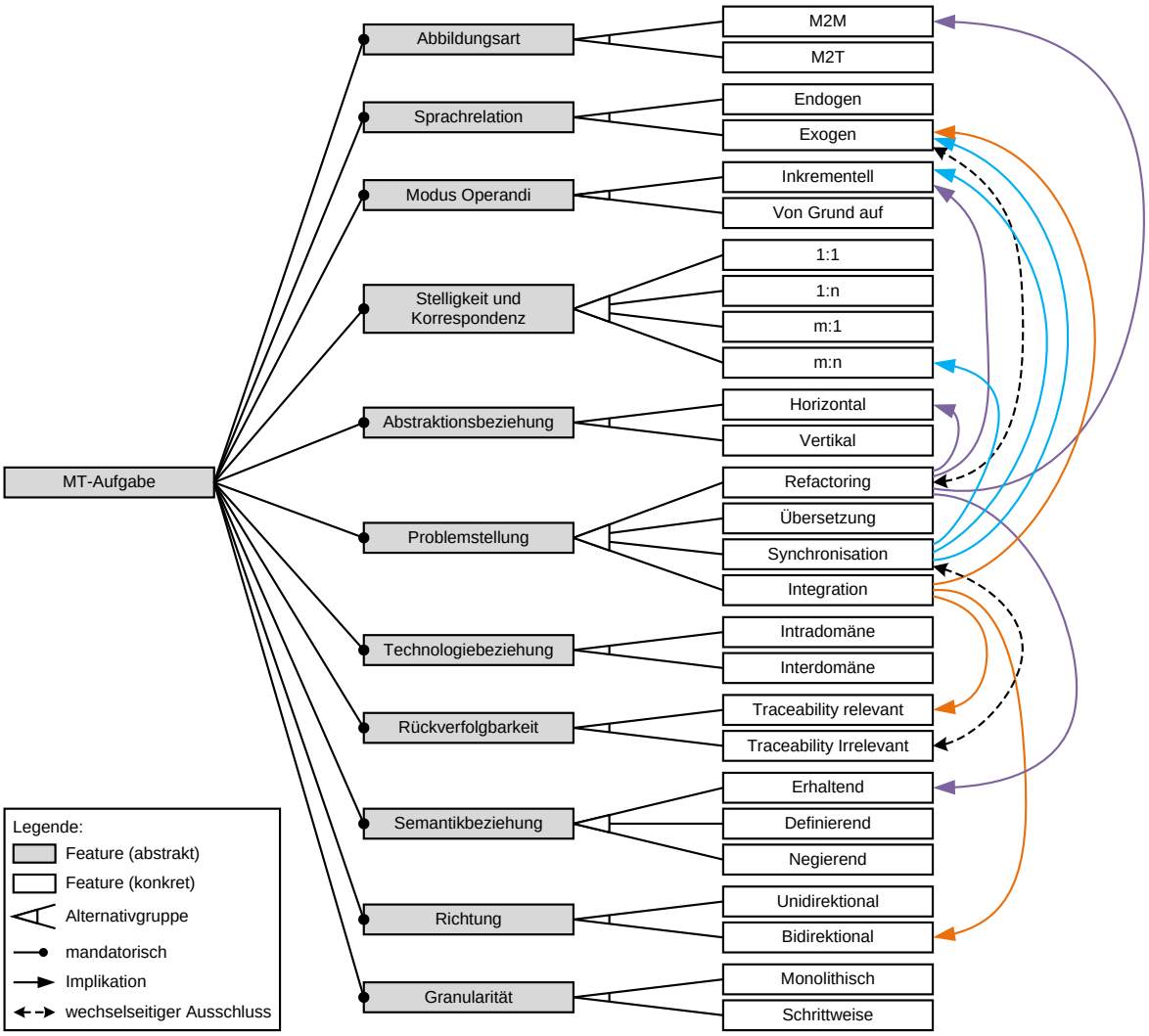


Abbildung 3.1: Übersicht wichtiger Aspekte einer Modelltransformationsaufgabe (in Anlehnung an die Art der Darstellung aus [CH06] sowie inhaltlich an [CH06; CH03; MV06; MCV05; REP12; Kus+13; WAS14]. Die unterschiedlichen Farben der Implikationspfeile dienen der besseren Lesbarkeit)

In Abbildung 3.1 sind wichtige Eigenschaften einer *MT*-Aufgabe sowie einige der naheliegenden Abhängigkeiten in Form eines *Feature-Modells*, vgl. [Kan+90], erfasst. Die Darstellung ist hierbei keinesfalls als erschöpfend bzw. vollständig anzusehen. Über die eingezeichneten Abhängigkeiten hinaus sind weitere Zusammenhänge möglich. Für eine ausführliche Darstellung der konkreten Syntax und der Semantik von Feature-Modellen sowie für Anwendungsbeispiele sei hier stellvertretend auf [Sch+07] und [BSR10] verwiesen. Im Folgenden erkläre ich Feature-Modelle kurz, um ein grundlegendes Verständnis solcher Modelle, vgl. Abbildung 3.1, zu ermöglichen.

[6] Verwendung unterschiedlicher *Meta-Metamodelle* (vgl. [MV06]).

Feature-Modelle sind hierarchische, baumartige Strukturen, deren Knoten mögliche Features (im Sinne von Eigenschaften) des modellierten Zusammenhangs oder Systems repräsentieren. Einfache Kanten in Feature-Modellen setzen Eltern- und einzelne oder Gruppen von Kind-Features in Abhängigkeitsbeziehung zueinander und schränken dadurch den Raum gültiger Feature-Konfigurationen (im Sinne einer „Auswahl von Features“) ein. Kind-Features können grundsätzlich nur dann ausgewählt bzw. in der Konfiguration „vorhanden“ sein, wenn auch ihre Eltern-Features ausgewählt sind. Darüber hinaus können weitere Abhängigkeiten zwischen Features existieren, insbesondere auch gerade dann, wenn diese nicht über eine direkte oder transitive Eltern-Kind-Beziehung verbunden sind. Abhängigkeiten dieser Art werden häufig als *Cross-Tree-Constraints* bezeichnet [BSR10]. Im Fall von Abbildung 3.1 lassen sich Abhängigkeiten zweier hier berücksichtigter Arten erkennen; sowohl *Implikationskanten* (im Sinne von $x \rightarrow y$) als auch Kanten für einen *wechselseitigen Ausschluss* (im Sinne von $\neg(x \wedge y)$) sind vorhanden. Für die konkrete Syntax vgl. die Legende in Abbildung 3.1.

Kind-Features können *verpflichtend* (engl. mandatory) sein, was bedeutet, dass die Auswahl des entsprechenden Eltern-Features auch die Auswahl des Kind-Features zwingend erfordert. Die Implikationskante besitzt eine vergleichbare Semantik, verbindet allerdings typischerweise keine Features, für die eine direkte Eltern-Kind-Beziehung besteht. Eine *Alternativgruppe* wird verwendet, um auszudrücken, dass aus der Gruppe der beinhalteten Kind-Features bei Auswahl des Eltern-Features genau eines ausgewählt werden muss (im Sinne von $p \rightarrow ((c_1 \wedge \neg c_2 \wedge \neg c_3 \wedge \ldots) \vee (\neg c_1 \wedge c_2 \wedge \neg c_3 \wedge \ldots) \vee \ldots)$.

Mit einem Feature-Modell lässt sich die *Variabilität* in einem System oder in einer Domäne systematisch erfassen. Jede valide widerspruchsfreie *Konfiguration* (vollständige Auswahl von Features) beschreibt eine mögliche Ausprägung, welche auch als *Produkt* bezeichnet wird.

Kommen wir auf das konkrete Feature-Modell aus Abbildung 3.1 zurück. Die zu lösenden Modelltransformationsaufgabe – hier repräsentiert durch das Wurzelelement *MT-Aufgabe* – ist durch elf wesentliche Eigenschaften charakterisiert, welche als *abstrakte* Features modelliert wurden.[7] So ist die Art der Abbildung, bezogen auf das übergreifende Ein- und Ausgabeverhalten der Transformation, entweder vom Typ *M2M* oder vom Typ *M2T*. Das Feature *Sprachrelation* beschreibt den sprachlichen Zusammenhang zwischen den Ein- und Ausgabemodellen. Falls die Ein- mit der Ausgabesprache übereinstimmt, ist die Transformation *endogen*, ansonsten *exogen*. Der *Modus Operandi* charakterisiert die Art der Ausführung, wobei als Optionen die *inkrementelle* Ausführung, bei der nur kleinere Veränderungen und zum Teil an bestehenden Artefakten auf Basis von *Deltas* vorgenommen werden, sowie die vollständige (Neu-)Ausführung im sogenannten *Batch*-Modus, bei der eine Lösung von Grund auf neu konstruiert wird, möglich sind. Die *Stelligkeit und Korrespondenz* beschreibt, welche Anzahl an Modellen auf Quellseite mit welcher Anzahl an Modellen auf Zielseite verknüpft wird. Der Standardfall ist die Option *1:1*, bei der ein Eingabemodell auf ein Zielmodell (oder Text) abgebildet wird. Das Feature *Abstraktionsbeziehung* beschreibt mit seinen beiden Kind-Features *Horizontal* und *Vertikal* den Zusammenhang zwischen Quell- und Zielseite der Transformation im Hinblick auf den Abstraktionsgrad. So sind Programme, bei denen *PIMs* auf *PSMs* abgebildet werden, Beispiele für vertikale Transformationen. Bei horizontalen Transformationen ändert sich dagegen das Abstraktionsniveau nicht.

[7] Demzufolge sind konkrete Kind-Features auszuwählen.

Bezüglich der zu lösenden *Problemstellung*, die einer *MT*-Entwicklung zugrunde liegt, sind, wie weiter oben bereits erwähnt, vgl. [LC08], verschiedene Optionen denkbar. In Abbildung 3.1 sind folgende Fälle berücksichtigt (die sich allerdings nicht eins zu eins mit den erwähnten Szenarien decken): (i) *Refactoring*, (ii) *Übersetzung*, (iii) *Synchronisation* und (iv) *Integration*. Da bei einem *Refactoring* das Quellmodell semantikerhaltend und nur begrenzt verändert wird, ist eine solche Abbildung als horizontal, inkrementell und endogen anzusehen. Diese Eigenschaften werden (nur) von *M2M*-Abbildungen erfüllt. Eine *Übersetzung* stellt dagegen den typischen Fall für Modelltransformationen dar, bei der Eingabemodelle oder Teile davon auf etwas Neues abgebildet werden. Von einer *Synchronisation* soll dann die Rede sein, wenn die Eingabe aus einer *Menge* von miteinander in Beziehung stehenden Modellen besteht, zwischen denen eine Synchronisationsabbildung definiert ist. Dabei ist der Zustand vor der Transformation als nur unvollständig umgesetzt anzusehen. Dies bedeutet, dass solche Modellelemente in mindestens einem der Eingabemodelle existieren, für die noch keine Entsprechung in den übrigen Modellen identifiziert wurde. Die Transformation stellt diesen Zusammenhang, wenn möglich, her. Der Zusammenhang zwischen den beteiligten Modellen kann dabei durch ein sog. *Traceability*-Modell beschrieben werden, welches dann im Rahmen der Synchronisation entweder von Grund auf neu aufgebaut oder ergänzt wird. Eine *Integration* stellt dagegen sicher, dass mehrere Modelle durch die Erzeugung noch fehlender Elemente auf einer der oder beiden Seiten so konsolidiert werden, dass sie zueinander konsistent sind (in Bezug auf eine gegebene Abbildungsvorschrift). Änderungen an einzelnen Modellen können so zu Änderungen an den übrigen Modellen führen. Dies bedeutet, dass diese Art der Transformation typischerweise exogen und bi- bzw. multidirektional ist. Synchronisation und Integration setzen eine Analyse der Zusammenhänge zwischen den beteiligten Modellen voraus, weshalb bei der *MT*-Aufgabe auch Aspekte der Rückverfolgbarkeit (engl. Traceability) relevant sind.

Das Feature *Technologiebeziehung* charakterisiert die *MT*-Aufgabe dahingehend, ob technologisch verschiedenartige Domänen überbrückt werden.[8] Im Falle von *Intradomänen*-Abbildungen beziehen sich die durch die Metamodelle beschriebenen Modellierungssprachen auf dieselbe technische Domäne. Ein Beispiel hierfür wären Abbildungen zwischen UML-Diagrammen. Bei *Interdomänen*-Abbildungen werden dagegen unterschiedliche „Modellierungswelten“ miteinander verknüpft (man denke z. B. an das Standard-Beispiel für Transformationen, die Abbildung von Klassendiagrammen auf Datenbankschemata [Béz+06b]).

Mittels des Features *Rückverfolgbarkeit* lässt sich charakterisieren, inwiefern Übersetzungsvorgänge nachträglich analysiert oder Abhängigkeiten zwischen Modellelementen für andere Arbeitsschritte ausgenutzt werden können. Im Modell aus der Abbildung wird dabei nur grob unterteilt: in Aufgaben bei denen Rückverfolgbarkeit entscheidend ist und Aufgaben bei denen sie irrelevant ist.

Mit dem Feature *Semantikbeziehung* wird die Ein- und Ausgabebeziehung auf der semantischen Ebene klassifiziert. Konkret gibt es drei Optionen, nämlich (i) semantikerhaltende Transformationen, (ii) semantikdefinierende Abbildungen und (iii) Abbildungen überwiegend syntaktischer Art, ohne unmittelbar erkennbaren Bezug zur Semantik.

Des Weiteren wird eine *MT*-Aufgabe durch die *Richtung* der Abbildung(en) gekennzeichnet. Typischerweise sind *MTs* gerichtet (von Quell- zur Zielseite) und *unidirektional*.

[8] Domäne ist hier im Sinne von *Technological Space* [KBA02] zu verstehen.

Allerdings kann eine Abbildung in entgegengesetzter Rückrichtung auch relevant sein (*bidirektionales* Transformationsproblem). Beispielsweise lassen sich manche *QVT*-Regeln in beide Richtungen interpretieren.

Als letzte Eigenschaft ist die *Granularität* einer *MT*-Aufgabe genannt. Einige Abbildungsarten sind durch ein schrittweises Vorgehen mittels Teiltransformationen gekennzeichnet, andere dagegen sind atomare, *monolithische* Prozesse ohne Zwischenergebnisse.

3.3 Modellverwaltung und Werkzeugunterstützung

Für die Ausführung einer Transformation ist es notwendig, dass die zu transformierenden Modelle in einer geeigneten Datenstruktur und Repräsentation vorliegen. Ein entsprechender dedizierter Speicher für Modelle wird typischerweise als Modell-Repository bezeichnet (vgl. hierzu auch die Ausführungen zu *XMI* aus Abschnitt 2.2.1). Ein gewisser Anteil zu transformierender Modelle entsteht in *proprietären* Modellierungswerkzeugen, was die Verfügbarkeit für *MTs* potentiell einschränken kann. Nach Sendall und Kozaczynski gibt es drei Möglichkeiten zum Zugriff auf entsprechende Modelle [SK03]:

1. Ein *direkter Zugriff auf die (interne) Modellrepräsentation des Werkzeugs* durch eine *Programmierschnittstelle/API*. Dies geschieht i. d. R.unter Rückgriff auf eine Standardprogrammiersprache (Java, C#, etc.) oder mittels Datenbankabfragesprachen (z. B. *SQL*). Die Autoren von [SK03] sehen bei diesem Ansatz Vorteile durch einen vergleichsweise geringen Einarbeitungsaufwand sowie eine hohe Akzeptanz unter den Entwicklern. Nachteilig können sich dagegen die Abhängigkeit von einer möglicherweise zu stark eingeschränkten *API* sowie der geringe Abstraktionsgrad auswirken.

   Ein Beispiel für diese Art des Zugriffs wären *MTs* zur Optimierung von [Stü+07b] bzw. zur Codegenerierung[9] aus Matlab-Simulink-Modellen. Im nächsten Kapitel wird eine Beispieltransformation aus diesem Kontext vorgestellt, allerdings ohne auf die Details des Werkzeugzugriffs einzugehen.[10]

2. Ein *Export der Modelle in ein offenes Austauschformat*, z. B. *XMI*. Eine Modelltransformation kann bei *XML*-basierten Dateiformaten sogar direkt auf dieser Darstellung mittels *XSLT* erfolgen, vgl. auch [SK03]. Einen Nachteil sehen Sendall und Kozaczynski allerdings in der Einschränkung auf *Batch*-Transformationen. Anders als bei inkrementellen Ansätzen, vgl. hierzu z. B. [Kus+13], gehen eventuell vorhandene Informationsstände auf der Zielseite verloren. Als weiteren Kritikpunkt führen Sendall und Kozaczynski die spezifischen Eigenheiten der *XSLT*-Sprache auf, welche die Sprache „unhandlich“ erscheinen lassen.[11]

[9] Vgl. hierzu: *Embedded Coder* von The MathWorks oder *TargetLink* von dSPACE.

[10] Die Wahl dieses Beispiels ist der Tatsache geschuldet, dass ich mich während meiner Anstellung an der TU-Darmstadt im Rahmen des BMBF-geförderten Forschungsprojekts ***Ma**tlab (Simulink und Stateflow) **J**ava **A**dapter (MAJA)* (Förderprogramm „IKT 2020 - Forschung für Innovationen“, Förderkennzeichen: `01 IS 09018 B`) unter anderem mit sogenannten *Online-Werkzeugadaptern* beschäftigen konnte. Letztere stellen eine intelligente Zwischenschicht dar (incl. Verzögern/Sammeln von Zugriffen, Sicherstellung konsistenter Sicht auf die Daten, etc.), die den Werkzeugzugriff für den Anwender transparent erscheinen lässt.

[11] Meine persönlichen Erfahrung mit *XSLT* decken sich mit dieser Aussage.

Ein Beispiel für eine komplexere *XSLT*-basierte Modelltransformation ist der Compiler bzw. Codegenerator namens ***MO****F Meta-****Mo****del* ***C****ompiler (MOMoC)*, vgl. [Bic03], der eine wichtige Komponente des am Fachgebiet Echtzeitsysteme entwickelten MOFLON-Werkzeugs, vgl. [Ame+06], darstellt.

3. Eine *direkte Unterstützung von MTs durch das Modellierungswerkzeug selbst.* In diesem Fall bietet das Modellierungswerkzeug selbst eine fest eingebaute Art der Abbildung oder eine domänenspezifische Sprache zur Spezifikation entsprechender Mappings an. Die wichtigste Eigenschaft einer solchen werkzeugspezifischen Sprache ist, dass sie ausreichend ausdrucksmächtig für die zu lösende Aufgabe ist [SK03]. Ansonsten sind selbstredend auch die in Abschnitt 3.1 erwähnten Eigenschaften potentiell relevant.

3.4 M2X im Detail

Wie bereits in Definition 3.1 aufgeführt, werden zwei Hauptklassen von Modelltransformationen unterschieden: *M2M*- und *M2T*-Transformationen. *M2M*-Transformationen stellen im *MDSD*-Umfeld, mit Modellen als primär genutzten Entwicklungsartefakten, den typischen Fall dar. *M2T*-Transformationen (z. B. Code- oder Dokumentengenerierung) sowie die eher selten als Modelltransformation bezeichnete inverse Text-zu-Modell-Abbildung (im Sinne von *Parsing* [WM97]) bilden die Brücke zwischen der *MDSD*-Welt und der Welt klassischer, textueller Sprachen. Letztere werden in diesem Zusammenhang häufig als *Grammarware* bezeichnet, vgl. z. B. [Fav04; KLV05]. Im Folgenden werden die beide *MT*-Arten dediziert betrachtet.

3.4.1 Modell-zu-Text

Sowohl in der klassischen als auch in der modellbasierten Softwareentwicklung sind textbasierte Artefakte von zentraler Bedeutung (man denke beispielsweise an *XML*). Formale Spezifikationen (vgl. hierzu z. B. [Lam00]), Dokumentationen, Quellcode, Testprotokolle und diverse Serialisierungsformate sind i. d. R.von textueller Gestalt. Sollen solche Artefakte aus Modellen generiert werden, spricht man von *M2T*-Transformationen im Allgemeinen bzw. von Codegenerierung im Speziellen. Das automatisierte Generieren von Text ist selbstverständlich viel älter als *MDSD*-Ansätze. So wurde bereits bei den frühesten Arten der Programmierung durch den Assembler Maschinencode aus der Assemblersprache generiert. Und die automatisierte Generierung von Dokumenten (Reports, Abrechnungen, Stücklisten, Serienbriefe etc.) war und ist eine der zentralen Aufgabe der *EDV*. Entsprechend angepasste Rahmenwerke und Sprachen, mit denen solche Aufgaben im Zusammenspiel mit Modellen gelöst werden können, sind augenscheinlich auch Bestandteil aller ernsthaften Ansätze zur modellbasierten Entwicklung.

Die technische Umsetzung von *M2T* kann auf verschiedene Arten erfolgen. Der Ansatz mit der größten Verbreitung dürfte der auf *Templates* (Dokumentvorlagen, Textschablonen) aufbauende sein. Hierbei werden unveränderliche Teile des zu erzeugenden Textartefakts in Form eines „Lückentextes“ vorgegeben. Letzterer wird direkt entwickelt oder aus konkreten Beispielen abgeleitet. Lücken, die im Template durch spezielle Auszeichnungen in der jeweiligen Template-Sprache markiert sind, werden bei einer Auswertung dyna-

misch mit konkreten Werten anhand des Modellinhalts gefüllt. Das fertige Textartefakt ergibt sich aus der Kombination der statischen und dynamischen Teile.

Im Java-Umfeld aktuell oft genutzte Template-Sprachen (ohne unmittelbaren Bezug zur Modellierung) sind z. B. *Velocity*[12] und *StringTemplate*.[13] Darüber hinaus bietet das Eclipse *M2T*-Projekt mit seinen Subprojekten (i) *JET*, (ii) *Xpand*, und (iii) *Acceleo* ebenfalls Optionen für Java- bzw. *EMF*-basierte Projekte mit direktem Bezug zu *MDSD*. Mit der *MOF Model to Text Transformation Language (MOFM2T)* [08] gibt es auch einen *OMG*-Standard zur Abbildung *MOF*-konformer Modelle auf Text.

Eine weitere Möglichkeit – neben der Ad-Hoc-Variante mittels `print-` bzw. `printf`-äquivalenten Anweisungen – ist der Rückgriff auf das *Besucher* bzw. *Visitor*-Entwurfsmuster von Gamma et al. [Gam+95]. Für die Details sei hier auf die Literatur verwiesen. Teile einer *M2T*-Abbildung lassen sich unter Umständen auch als *M2M*-Transformation(en) auffassen, wie beispielsweise von Wimmer und Burgueño in [WB13] ausgenutzt.

3.4.2 Modell-zu-Modell

In Abbildung 3.2 ist der Zusammenhang zwischen Quellmetamodell ($\mathsf{MM}_{\mathsf{Src}}$) und Zielmetamodell ($\mathsf{MM}_{\mathsf{Trgt}}$) sowie den dazu konformen Modellen $\mathsf{Model}_{\mathsf{In}}$ (Quelle) und $\mathsf{Model}_{\mathsf{Out}}$ (Ziel) für den Fall einer *M2M*-Transformation skizziert. Die Modelltransformation im Sinne von Punkt (i) der Definition 3.1, s. die beiden schraffierten Blöcke, definiert diesen Zusammenhang. Entsprechende Transformations*regeln* sind unter direkter Bezugnahme auf die Elemente der Quell- und Zielmetamodelle formuliert, vgl. die «references»-Kanten, welche von den Transformationsregeln ausgehen. Die Regeln können ggf. selbst als Modelle aufgefasst werden, konform zu einem Transformationsmetamodell ($\mathsf{MM}_{\mathsf{MT}}$). Tatsächlich ausgeführt werden die Regeln mit Hilfe der *Transformationsmaschine* (engl. *Engine*). Dazu wird eine ausführbare *Operationalisierung* der Regeln benötigt. Die grau hinterlegten Elemente sowie die «executes»-Kante repräsentieren den *MT*-Prozess entsprechend der zweiten Interpretation des Transformationsbegriffs aus Definition 3.1.

In Abbildung 3.2 ist ein weiterer Aspekt angedeutet: Unter gewissen Umständen stellen die (regelbasierten) Beschreibungen der Abbildungsvorschrift zwischen Quell- und Zielmetamodell bereits selbst die ausführbare *Implementierung* der Transformation dar (vgl. hierzu Informationen zu deklarativen, regelbasierten *MT*-Sprachen). Aus diesem Grund ist die Menge der Regeln in der Abbildung ebenfalls schraffiert dargestellt. Aber selbst in diesem Fall liegt die Existenz einer abstrakteren, nicht unmittelbar (z. B. durch einen Interpreter) ausführbaren *Spezifikation* der Transformationsaufgabe nahe, denn die Transformationsregeln entstehen gewöhnlich nicht einfach „aus dem Nichts heraus“. Zumindest der Entwickler besitzt eine mentale Vorstellung von ihnen. Diese Vorstellung kann als implizite, informelle Transformationsspezifikation gelten, dann allerdings ohne Bezug zu einem entsprechenden Metamodell, weshalb dieses in Abbildung 3.2 nur in Grau angedeutet ist. Im Falle von Transformationssprachen, die nicht regelbasiert sind, würde in der Abbildung die Transformationsspezifikation an die Stelle der Transformationsregeln treten. Die Berücksichtigung von Spezifikation und Regeln in der Abbildung soll verdeutlichen, dass Transformationsregeln von Fall zu Fall mal der Implementierung (hier dargestellt) und mal der Problembeschreibung zuzurechnen sind.

[12] The Apache Velocity Project, `http://velocity.apache.org/` (abgerufen am 6.11.2013).

[13] Teil des ANTLR-Frameworks von T. Parr, `http://www.stringtemplate.org/` (abgerufen am 6.11.2013).

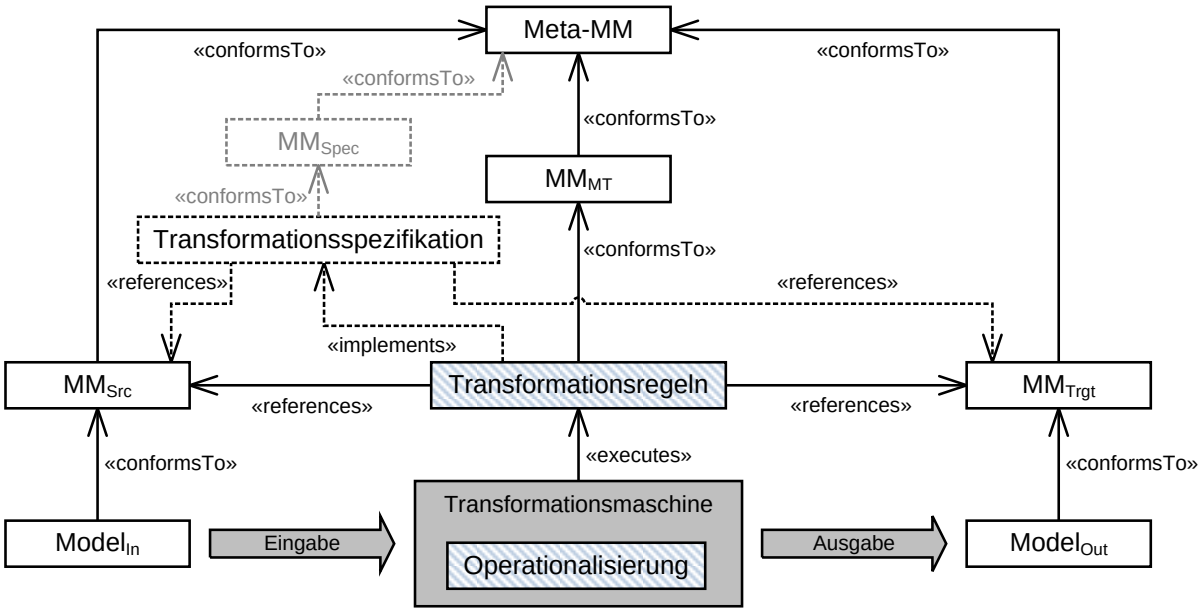


Abbildung 3.2: Zusammenhänge zwischen Modelltransformationen und Metamodellen (in Anlehnung an [REP12, Abb. 2] sowie [Béz+06a] und [KWB03, Abb. 2-7, S. 26])

Einige der zuvor verwendeten Konzepte, die bisher noch nicht definiert wurden, werden im folgenden genauer gefasst. Insbesondere sind dies die Konzepte der *Transformationsregel* sowie deren *Operationalisierung*, deren Details im weiteren Verlauf wichtig sind. Am Anfang steht Definition 3.2, in der das *allgemeine* Konzept einer Transformationsregel festgehalten ist.

Definition 3.2 (Transformationsregel (allg.), angelehnt an [KWB03]):

Als Transformationsregel *bezeichnen wir im Folgenden die kleinstmögliche* deklarative Abbildungsvorschrift*, welche beschreibt, wie eine definierte, festgelegte (endliche) Menge von Konstrukten einer Quellsprache*[a] *auf eine definierte, festgelegte (endliche) Menge von Konstrukten einer Zielsprache*[a] *systematisch und atomar durch lokal*[b] *begrenzte Änderungen (im Sinne von Hinzufügen, Löschen, Modifizieren) abzubilden ist.*

[a] Jeweils definiert über Quell- respektive Zielmetamodell(e).
[b] vgl. [And+99, S. 2]

Definition 3.2, die beispielsweise auch auf sog. *QVT-Rules* (*Relations* oder *Mappings*) zutrifft [OMG11, S. 27], wird zusätzlich ergänzt durch die spezifischere Definition 3.3 für *spezielle* Transformationsregeln. Diese Definition ist besser geeignet, um die Zusammenhänge des nächsten Kapitels zu erfassen, da sie sich unter anderem explizit auf die Struktur und die Bestandteile von hier verwendeten Regeln bezieht. Im nächsten Kapitel folgt darüber hinaus noch eine weitere Verfeinerung, die sich explizit auf sog. Graphtransformationsregeln bezieht.

Definition 3.3 (Transformationsregel (speziell), in Anlehnung an [Ehr+06, Kap. 1.2.1]):

Eine Transformationsregel umfasst eine Linke Seite *bzw. Left Hand Side (LHS), L, welche die* Vorbedingungen *zur Anwendbarkeit der Regel festlegt, und eine* Rechte Seite *bzw. Right Hand Side (RHS), R, welche die* Nachbedingungen *der Regel beschreibt. Darüber hinaus bestimmt die Regeldefinition auch die sog.* Schnittstelle *(engl. Interface), K. Letztere besteht nur aus den Elementen der „Überlappung" von LHS und RHS: $K \sqsubseteq L \sqcap R$ (in Anlehnung[a] an die Mengenoperatoren $\subseteq$ und $\cap$).*

Alle Elemente, die ausschließlich in der LHS der Regel auftreten, also in $L - K$ enthalten sind, entsprechen den vor der Regelanwendung existierenden Entitäten des Modells, welche durch die Regelanwendung gelöscht werden.

Alle Elemente, die ausschließlich in der RHS der Regel auftreten, also in $R - K$ enthalte sind, entsprechen den durch eine Regelanwendung anzulegenden Entitäten im Modell.

[a] Da es sich bei L und R i. Allg. nicht um mathematische Mengen handelt, kann man hierbei nur informell von (Durch-)Schnitt sprechen.

Mit Definition 3.4 wird nun noch das fehlende Konzept der Operationalisierung einer Regel eingeführt. Auf technischer Ebene gibt es, wie bereits erwähnt, zwei Ansätze um aus einer regelbasierten, deklarativen Transformationsbeschreibung eine ausführbare Operation abzuleiten. Zum einen kann die Beschreibung in eine ausführbare Sprache *kompiliert* respektive *übersetzt* werden. Dabei werden statisch durchführbare Berechnungen vorweggenommen. Zum anderen kann die Beschreibung direkt durch einen Interpreter ausgeführt werden. Vergleiche hierzu beispielsweise [WM97] und auch die entsprechenden Aussagen aus Kapitel 1.2.

Definition 3.4 (Operationalisierung einer Regel):

Eine durch eine (Transformations-)Maschine direkt ausführbare Repräsentation einer deklarativen Regel, sprich ihre Implementierung,[a] wird im Folgenden als Operationalisierung *bezeichnet.*

Dabei umfasst die Operationalisierung einerseits die notwendigen Informationen für die einzelnen Arbeitsschritte zur Regelauswertung. *Im Einzelnen sind dies Informationen zu den notwendigen Schritten (i) zum Suchen und Finden einer Anwendungsstelle auf Basis des durch die LHS vorgegebenen Musters (engl. Pattern), im Folgenden als* Mustersuche *bzw.* Pattern Matching *bezeichnet, (ii) zum Entfernen der zu löschenden Elemente sowie (iii) zum Anlegen der neu zu erzeugenden Elemente. Die Durchführung aller schreibenden Schritte wird gemeinsam als* Ersetzungsschritt *bzw.* Rewriting *bezeichnet.*

[a] Vgl. hierzu [WM97, S. 2].

Zusätzlich nutzt die Operationalisierung i. d. R. *Hilfsdatenstrukturen und -algorithmen* im Rahmen der Regelauswertung. Diese können beispielsweise als Bestandteil des Generats oder als Teilfunktionalität des Interpreters vorkommen.

> The wide applicability of graph grammars is due to the fact that graphs are a natural way of describing complex situations on an intuitive level.
>
> *(G. Rozenberg, aus dem Vorwort von [Roz97])*

4 Graphtransformationen

Im vorigen Kapitel wurden Modelltransformationen als Eckpfeiler der modellbasierten Softwareentwicklung vorgestellt. Nun werden wir einen bestimmten Ansatz zur Beschreibung von *MTs* näher betrachten, nämlich die sog. *Graphtransformationen (GTs)*. Insbesondere für *M2M*-Transformationen wurden im letzten Kapitel bereits regelbasierte Beschreibungen eingeführt. Dieses Konzept ist auch weiterhin zentral, da es sich bei Graphtransformationen um eine inhärent regelbasierte Beschreibung von Ersetzungsschritten auf Graphen handelt.

Das Kapitel ist unterteilt in vier Abschnitte. Im ersten Abschnitt werden Graphtransformationen informell eingeführt. Zusätzlich werden Hinweise auf entsprechende Übersichtsarbeiten zur weiterführenden Vertiefung in die Thematik gegeben.

Im zweiten Abschnitt werden *GTs* unter theoretischen Gesichtspunkten betrachtet. Ergänzend dazu wird ein Überblick wichtiger Formalisierungsvarianten gegeben. Letztere können im Rahmen dieser Arbeit allerdings nicht abschließend und vollumfänglich dargestellt werden. Wichtig sind dagegen einige wenige Konzepte und Begrifflichkeiten, deren Kenntnis erst ein grundlegendes Verständnis ermöglicht. Herausforderungen aufgrund des problemimmanenten Nichtdeterminismus werden in diesem Zuge ebenfalls thematisiert.

In Abschnitt drei wird die für diese Arbeit zentrale Unterklasse der Graphtransformationen eingeführt, die sog. *programmierten Graphtransformationen*. Dies geschieht unter Bezugnahme auf die hier verwendete *MT*-Sprache. Deren Vorstellung beinhaltet einen Überblick der konkreten und abstrakten Syntax sowie eine informelle Beschreibung der Semantik der Sprache.

Der vierte Abschnitt ist schließlich der Vorstellung einer konkreten Transformation gewidmet. Diese dient als Anschauungsobjekt einerseits der Vorstellung der *GT*-Sprache und andererseits als Beispiel einer nichttrivialen Transformation im weiteren Verlauf der Arbeit.

4.1 Allgemeine Grundlagen

Viele Systeme (bzw. (innere) Zustände dieser) sowie diverse Modellarten weisen eine Struktur auf, die sich besonders gut in Form eines oder mehrerer *Graphen* beschreiben lässt [Roz97]. Ein Graph ist eine mathematische Struktur und umfasst eine Mengen von *Knoten* und *Kanten*. Knoten können mittels Kanten verbunden werden (Kanten enden stets an Knoten). Es existieren unterschiedliche Darstellungen und Auffassungen von Graphen, sog. Graphmodelle, mit leicht unterschiedlichen Eigenschaften. So bestehen beispielsweise Unterschiede dahingehend, ob parallele Kanten (Mehrfachkanten) erlaubt sind, ob Kanten eine Richtung aufweisen (gerichtete Graphen) oder ob Kanten stets genau zwei Enden oder aber beliebig viele Enden (Hypergraphen) besitzen. Darüber hinaus können Knoten (und Kanten) mit einfachen Symbolen (beschrifteter Graph) oder komplexeren Attributen[1] (attributierter Graph) versehen sein [Roz97; Ehr+06].

In der theoretischen Informatik bildete sich ab der zweiten Hälfte der 1960er Jahre ein Teilgebiet heraus, in dem sich Forscher mit dem *regelbasierten Aufbau* und der *Analyse von Graphen* – jeweils auf Grundlage sog. *Graphgrammatiken*[2] – sowie der regelbasierten Modifikation von Graphen, den sog. *Graphtransformationen*, beschäftigten. Für eine sehr umfassende Abhandlung über dieses Gebiet sei auf das dreibändige „Handbuch“ verwiesen [Roz97; Ehr+99a; Ehr+99b]. Es geht auf diverse Strömungen, auf Anwendungsbeispiele und (mittlerweile zum Teil als historisch anzusehende) Werkzeuge sowie auf spezifische und fundamentale Aspekte der Theorie ein.

Seit der Etablierung dieses Teilgebiets wurden und werden Graphtransformationen in vielen unterschiedlichen Anwendungsdomänen gewinnbringend eingesetzt. Andries et al. führen in [And+99] beispielsweise Beiträge zur Mustererkennung, zu der Beschreibung der Semantik von Programmiersprachen, zu der Implementierung von Compilern oder zu der Spezifikation von Datenbanken und verteilten Systemen auf. Insbesondere im Rahmen der Definition von Syntax und Semantik sog. *Diagrammsprachen* sind Graphtransformationen besonders nützlich [BH02]. Für Spezifikationsaufgaben lassen sie sich beispielsweise auf zwei unterschiedlichen Arten verwenden, wie in [Ren10] dargelegt:

1. Graphtransformationen können selbst als *Modellierungssprache* zur unmittelbaren Beschreibung von Systemverhalten genutzt werden. (Die vorliegende Arbeit folgt überwiegend dieser Interpretation.)
2. Die Semantik einer Modellierungssprache kann auf eine „Graphtransformationssemantik“ abgebildet und hierdurch definiert werden. Beispielsweise lässt sich das Verhalten konkreter Petri-Netze mit Hilfe von Graphtransformationen beschreiben, vgl. hierzu z. B. [Kre81].

4.1.1 Grammatiken über Graphen

Im Falle von *Graphgrammatiken* lassen sich sog. *Graphersetzungsregeln* als *Produktionen* einer auf Graphen verallgemeinerten Form von *formaler Grammatik* interpretieren.[3] Durch eine endliche und bekannte Menge an Produktionen wird eine formale Sprache „aufgespannt“, deren (ggf. unendlich große) Wortmenge aus (endlichen) Graphen be-

[1] Attribute lassen sich anschaulich als Schlüssel-Wert-Paare auffassen.
[2] Auch als *Web-Grammars* [PR69] bezeichnet, vgl. [Roz97, Kap. 7.1, S. 481].
[3] Solche Graphgrammatiken wurden in einer grundlegenden Arbeit von Pfaltz und Rosenfeld [PR69] eingeführt.

steht. Im Hinblick auf die Beschreibung von Sprachen mit graphartiger Struktur sind Graphgrammatiken und Metamodelle vergleichbar, wenn auch fundamental verschieden. Erstere sind konstruktiv, letztere dagegen beschreibend bzw. deskriptiv [EKT09].

Die Zusammenhänge und Übersetzbarkeit beider Beschreibungsarten ist in jüngerer Zeit Gegenstand aktiver Forschung, vgl. z. B. [EKT09; Win+08; Tae12]. Dabei wird versucht, eine allgemeingültige Konstruktionsvorschrift zu entwickeln, um aus einem (geringfügig eingeschränkten) Metamodell eine Graphgrammatik so abzuleiten, dass für jede Modellinstanz des Metamodells eine gültige Ableitung für die Grammatik existiert (im Sinne der *Vollständigkeit*), und dass jeder (durch die Grammatik) ableitbare Graph auch eine Instanz des Metamodells ist (im Sinne der *Korrektheit*).

Zusammenfassend lässt sich festhalten, dass für Graphgrammatiken zwei Hauptanwendungsfälle bestehen. Zum einen die Analyse, ob ein gegebener Graph Teil einer gegebenen Sprache ist, also dem *Parsen von Graphen*. Zu anderen *die Generierung* bzw. *Ableitung konkreter Graphen*. Beide Aspekte können für das Testen von *MTs* und *GTs* Relevanz besitzen, wie in Kapitel 6 genauer dargelegt.

4.1.2 Transformationen von Graphen

Reine *GTs* beschreiben auf *deklarative* Art eine *schrittweise, regelbasierte Modifikation* von Graphen. Dabei ist der initiale Graph im Gegensatz zum Grammatik-Fall i. Allg. nicht leer, sondern wird vorgegeben. Auch muss es sich bei einem Transformationsprozess, im Gegensatz zum Ableitungsprozess bei Graphgrammatiken, nicht zwingend um eine *endliche* Sequenz von (Ersetzungs-)-Schritten handeln, da Endlosschleifen bzw. endlose Rekursionen grundsätzlich möglich sind. In der Praxis sind terminierende Transformationen allerdings viel sinnvoller.

Eine *Graphtransformationsregel* umfasst, analog zu Definition 3.3, eine *Linke Seite* (*LHS*) und eine *Rechte Seite* (*RHS*). Auch bei *GTs* definiert die *LHS* der Regel, welche Vorbedingungen zur Anwendung der Regel erfüllt sein müssen. Die *RHS* der Regel bestimmt dagegen die Nachbedingungen nach der Anwendung der Regel. Die eigentliche Modifikation des Graphen durch die Regelanwendung wird als *Rewriting* bezeichnet. Im Zuge dessen werden Teile eines *Instanzgraphen* (engl. *Host Graph*), für den eine betrachtete Regel ausgewertet werden soll, durch entsprechende Elementen der *LHS* der Regel identifiziert. Zur Auswertung einer Regel ist es folglich notwendig, die entsprechenden Teile des Instanzgraphen auf Grundlage der *LHS* per *Mustersuche* respektive *Pattern Matching* zu bestimmen, vgl. hierzu auch Definition 3.4.

Wurde eine *Anwendungsstelle* bzw. ein *Treffer* (engl. *Match*) für die Regel gefunden, so ist die Regel *anwendbar* (falls keine anderweitigen Anwendungsbedingungen dies ausschließen); die Regel passt also grundsätzlich zu der identifizierten Stelle im Graphen. Das Anwenden der Regel impliziert, dass alle Elemente des Instanzgraphen, die durch die Elemente der *LHS* identifiziert werden, aus dem Instanzgraphen entfernt werden und an ihrer Stelle eine (isomorphe) Kopie der *RHS* der Regel in den Instanzgraphen eingebaut wird [Roz97, S. 3]. Diese Darstellung ist vereinfacht, da i. d. R. zwar Elemente durch die *LHS* vorausgesetzt werden, diese aber im Rahmen der Regelanwendung nicht immer zwingend alle entfernt werden müssen. Ein Löschen gefolgt von einem anschließenden Neuanlegen wäre beispielsweise sehr ineffizient für zu konservierende Elemente, was nahelegt, dass in Umsetzungen eben nicht alle Elemente der *LHS* immer entfernt und alle Elemente der *RHS* immer neu erzeugt werden.

Die Formalisierung der Begriffe hat Einfluss sowohl auf die Mustersuche als auch das Rewriting. Beispielsweise sind die Formalisierungen der Abbildungen zwischen den Elementen von *LHS* und *RHS* zur Beschreibung nicht zu löschender Elemente, der Art der Abbildung des Mustergraphen[4] in den Instanzgraphen sowie der Details der *Einbettung* der Kopie der *RHS*[5] in den Instanz- bzw. Zielgraphen von entscheidender Bedeutung für Ausdrucksmächtigkeit und Komplexität des GT-Ansatzes, vgl. [Roz97]. Bezogen auf den *Einbettungsmechanismus* gibt es beispielsweise zwei grundsätzliche Herangehensweisen, wie in [Roz97, S. 3] dargelegt:

1. *Verkleben* nach dem *Gluing Approach*, auch als *Algebraischer Ansatz* bekannt [Roz97, S. 3], bei dem Elemente der *LHS* und auch der *RHS* der Regel unmittelbar auf Elemente des Zielgraphen abgebildet werden.[6]
2. *Neuverbinden* nach dem *Connecting Approach*, auch als algorithmischer oder mengentheoretischer Ansatz bekannt [Roz97, S. 3], bei dem alle Bildelemente der *LHS* sowie „Brückenkanten“[7] vollständig aus dem Zielgraphen entfernt werden und die Kopie der *RHS* mit dem restlichen Teil des Zielgraphen neu verbunden werden muss. Dazu werden neue Kanten benötigt, deren Erzeugung beispielsweise unter Bezugnahme auf die zuvor entfernten Brückenkanten erfolgen kann.

4.1.3 Hinweise zu weiterführender Literatur

Die kompakte Einführung in das Thema wird dem Umfang des Forschungsgebiets zu Graphtransformationen kaum gerecht. Deshalb folgen nun noch Hinweise auf einige umfangreichere Übersichtsarbeiten zur Thematik. Sie sollten geeignet sein, um eine noch umfassendere Intuition zu Graphtransformationen im Allgemeinen zu erhalten, bevor man sich ggf. dem Studium des Handbuches [Roz97] oder auch des aktuellen Standardwerks zu *algebraischen GTs* [Ehr+06] widmet.

So wurde in [And+99] der Versuch unternommen, die verschiedenen GT-Ansätze, die ebenfalls im nachfolgenden Abschnitt 4.2.1 kurz zusammenfassen sind, in ein gemeinsames Gedankengebäude einzusortieren. Auf konzeptioneller Ebene werden die Abgrenzungen zueinander verdeutlicht und daraus der Bedarf für ein Strukturierungskonzept (die sogenannten Transformation-Units) abgeleitet, welches unabhängig vom konkreten GT-Ansatz ist. Im Rahmen der Arbeit werden neben einer konzeptionellen Einführung in GT auch unterschiedliche Anwendungsszenarien vorgestellt.

Die Einführung [BH02] beschreibt Graphtransformationen anhand von Anwendungsfällen aus der Software-Engineering-Domäne und führt *GTs* auf Basis einer auf Mengen basierenden Interpretation des sog. *DPO*-Ansatzes ein. Letzterer wird im nachfolgenden Abschnitt 4.2.1 kurz thematisiert. Dazu wird in einer kompakten Darstellung auf diese am häufigsten genutzte Formalisierung eingegangen und auch auf entsprechende Alternativen hingewiesen. Die Autoren erklären ebenfalls, wie mit Hilfe von Graphtransformationen entweder konkrete Abbildungen innerhalb eines zu modellierenden Systems beschrieben werden können (Modeling), oder aber die Syntax und die Semantik von sogenannten Diagramm-Sprachen allgemein definiert werden kann (Meta-Modeling). Motiviert wird

[4] Vereinzelt auch als *Muttergraph* bezeichnet [Roz97, S. 5].
[5] Auch als *Tochtergraph* bezeichnet
[6] Man kann sich das bildlich so vorstellen, dass die Knoten und Kanten der beiden Regelgraphen auf die Knoten und Kanten des Zielgraphen deckungsgleich „geklebt“ werden.
[7] In Anlehnung an die Nomenklatur aus [Roz97, Abschnitt 1.1].

dies dadurch, dass (i) graphartige Strukturen von traditionellen Techniken zur Sprachdefinition (z. B. auf Basis von kontextfreien Grammatiken) nur unzureichend unterstützt werden, (ii) durch Graphtransformationen ein hoher Grad an Automatisierung und hohe Produktivität erreicht werden kann und (iii) Graphtransformationen ein probates Mittel zur Beschreibung und zur Definition von Semantiken graphartiger Sprachen darstellen.

In der Arbeit [Tae+05] untersuchen die Autoren Graphtransformationen im Hinblick auf Modelltransformationsaufgaben. Dazu werden vier bekannte Graphtransformationswerkzeuge[8] und die entsprechenden Ansätze miteinander verglichen, indem jeweils die gleiche Aufgabe gelöst wird. Als Referenzmaßstab dient *QVT*, das ebenfalls in den Vergleich aufgenommen wurde.

In [VSV05] werden ebenfalls *GT*-Werkzeuge verglichen, allerdings nicht in Bezug auf typische Werkzeug- und Spracheigenschaften, wie Benutzbarkeit oder Ausdrucksmächtigkeit, sondern im Hinblick auf Ausführungsgeschwindigkeit. Motivation hierfür ist die praktische Bedeutung der Frage bei der Auswahl eines Ansatzes sowie das Fehlen eines Benchmarks für den qualitativen Vergleich von Werkzeugen und Algorithmen. Ein weiterer wichtiger Beitrag ist die Klassifikation von *GT*-Problemstellungen bezüglich performanzrelevanter Charakteristika (z. B. Mustergröße oder dem durchschnittlichen *Fan-Out* von Knoten im Muster).

In [Hec06] werden *GTs* recht anschaulich und weitestgehend informell eingeführt. Darüber hinaus werden die folgenden erweiterten Konzepte vorgestellt und erklärt: (i) globale Nebenbedingungen (Constraints) auf Metamodellebene, um beispielsweise Einschränkungen durch Multiplizitäten zu modellieren, (ii) Multi-Objekten, um „allquantifizierte“ Regelauswertungen zu spezifizieren (iii) positive und negative Anwendungsbedingungen für einzelne Regeln – letztere werden auch als *Negative Application Conditions (NACs)* bezeichnet – sowie die (iv) durch einen Kontrollfluss gesteuerte Regelauswertung.

In [Ren10] werden verschiedenen Möglichkeiten zur formalen Beschreibung von Graphen zusammengefasst sowie die Vor- und Nachteile der jeweiligen Darstellungen verglichen. Darüber hinaus wird untersucht, wie gut sich Graphtransformationen als Formalismus zur Semantikbeschreibung und Modellierung eignen. Auch werden mögliche „Hemmschuhe“ untersucht, die einer weiteren Verbreitung von *GTs* entgegenstehen.

4.2 Theorie

Richten wir nun den Blick auf Aspekte von *GTs*, die deren reichhaltige Theorie betreffen bzw. sich aus dieser ergeben. Den Beginn macht eine Überblicksdarstellung der beiden wichtigsten Formalisierungsansätze, gefolgt von einigen Basisdefinitionen. Eine Betrachtung des immanenten Nichtdeterminismus, der das Testen entsprechender Transformationen erschweren kann, bildet den Abschluss dieses Abschnitts.

4.2.1 Formalisierungsvarianten im Überblick

Bei Graph*grammatiken* werden nach [Roz97] die beiden Ansätze (i) *Node Replacement Graph Grammars*, (ii) *Hyperedge Replacement Graph Grammars* unterschieden. Bezogen auf Graph*transformationen*, führen die jeweiligen Autoren in [Roz97] sowie [Ehr+06] unter anderem die folgenden Ansätze auf: (i) *Algebraische Beschreibung*, (ii) *Algorithmische*

[8] AGG, ATOM^3, VIATRA2 und VMTS

Beschreibung, (iii) Beschreibungen mittels monadischer (Prädikaten-)Logik zweiter Stufe, sowie (iv) Beschreibungen mittels sog. 2-Structures. Die beiden erstgenannten Ansätze stellen bei Transformationen die Hauptströmungen dar; sie bilden die theoretische Basis einer Vielzahl praktisch nutzbarer Werkzeuge und wichtiger Beweise. Weiter unten sind beide Ansätze kurz beschrieben.

In jüngerer Zeit hat der Algebraische Ansatz im akademischen Umfeld vermehrt an Verbreitung gewonnen. Er stellt, spätestens mit der Veröffentlichung von [Ehr+06], die zurzeit vorherrschende[9] Art der Formalisierung dar, und wird sowohl in der Lehre als auch in der Forschung vornehmlich verwendet.

Algebraische Graphtransformationen

Eng verbunden mit der Person und der Arbeitsgruppe von Professor Dr. H. Ehrig (Technische Universität Berlin) ist der Algebraische Ansatz, welcher in der mathematischen *Kategorientheorie* verwurzelt ist und sich auf deren Konzepte (Objekte, Morphismen, Funktoren etc.) und Konstruktionen (Pushout, Pullback, Limit, Co-Limit[10]) stützt. Ohne auf die genauen Details eingehen zu wollen, sei hier nur erwähnt, dass in diesem Ansatz beispielsweise Knoten und Kanten eines Graphen als Objekte im kategorientheoretischen Sinn aufgefasst werden. Einbettungen der Graphmuster in einen zu modifizierenden Graphen werden durch Morphismen beschrieben. Das Auswerten von GT-Regeln lässt sich auf die Konstruktion von Pushouts in der zur Aufgabenstellung passenden Kategorie zurückführen [Roz97; Ehr+06]. Diese Art der Formalisierung ermöglicht es, eindeutig festzulegen, welche Elemente des Graphen durch eine Regel wie zu modifizieren sind und welche Elemente ggf. wie neu angelegt werden müssen.

Der Algebraische Ansatz gliedert sich wiederum in zwei Interpretationen auf, wobei sich die erste Interpretation, welche unter der Bezeichnung „*DPO*-Ansatz“ (für *Double Pushout*) [EPS73; Cor+97] firmiert, als restriktiv bzw. *konservativ* (vgl. z. B. [Ren10; Gie+12]) umschreiben lässt. Bei ihr sind, per Definition, bestimmte Fälle ausgeschlossen, die in der zweiten Interpretation, gemeinhin unter der Bezeichnung „*SPO*-Ansatz“ (für *Single Pushout*) [Löw93; Ehr+97] bekannt, zugelassen sind. Bei letzterem Ansatz werden beispielsweise *hängende Kanten*[11] als Seiteneffekt implizit mit gelöscht [Ren10], die im anderen Ansatz entweder explizit behandelt werden müssten [Ren10] oder andernfalls eine Regelanwendung ausschließen würden [Gie+12]. Trifft der letztgenannte Fall ein, so spricht man von einer Verletzung der sog. *Dangling-Edge-Bedingung*, vgl. z. B. [Gie+12].

Bekannte Werkzeuge, die auf dem algebraischen Ansatz aufbauen bzw. diesen implementieren, sind beispielsweise (i) *AGG* [LB93; Tae00], (ii) *Henshin* [Erm+10; Bie+10b], (iii) *GrGen* [Gei+06; JBK10] sowie (iv) *GROOVE* [Ren04].

Algorithmische Graphtransformationen

Der Algorithmische Ansatz – historisch verbunden mit der Person und der Arbeitsgruppe von Professor Dr. M. Nagl (RWTH Aachen) – steht traditionell in einem stärke-

[9] Vgl. hierzu z. B. die Veröffentlichungen der einschlägigen ICGT-Konferenzreihe.

[10] Die entsprechenden deutschen Fachtermini sind: Fasersumme, Faserprodukt, Limes, Kolimes (vgl. z. B. [Par69, Kap. 2.6 und 2.7]).

[11] Auch als „baumelnde“ Kanten (engl. dangling edges) bezeichnet. Sie sind dadurch gekennzeichnet, dass sie selbst nicht Teil des Musters sind, ihnen aber durch die löschenden Effekte einer Regelanwendung (mindestens) einer ihrer Endknoten abhandenkommen würde.

ren Bezug zu einer unmittelbaren, praktischen Anwendbarkeit. Damit stehen vor allem auch Aspekte einer algorithmischen Durchführung von Graphtransformationen im Fokus, vgl. [Nag87]. Die Festlegung der Semantik entsprechender Transformationen kann dazu entweder mit Hilfe einer mengentheoretisch begründeten [Nag87] oder einer logisch begründeten [Sch91a; Sch97] Theorie erfolgen.

Das wahrscheinlich bekannteste Werkzeug, das auf den Algorithmischen Ansatz zurückzuführen ist, ist PROGRES [Sch91b; Sch91a; Zun98]. Es kann als archetypisches Beispiel für die Verwendung des Algorithmischen Ansatzes zur Semantikdefinition angesehen werden, die vor allem im Rahmen der Dissertation von Schürr (mittels mathematischer Logik) erfolgte, vgl. [Sch91a]. Die im Folgenden verwendete Graphtransformationssprache steht in gewisser Tradition zu PROGRES, unter anderem durch die bei der jeweiligen Entwicklung beteiligten Personen. In Abschnitt 4.3 wird dieser Zusammenhang noch einmal aufgegriffen und thematisiert.

Da die in dieser Arbeit verwendete und im Folgenden vorgestellte Transformationssprache de facto ebenfalls dem algorithmischen Paradigma zuzurechnen ist, sind die Theorie betreffenden Details des Algebraischen Ansatzes weitestgehend irrelevant für den weiteren Verlauf. Eine geeignete Formalisierungsvariante kann als zweitrangig für die vorgestellten Testansätze angesehen werden, da vorausgesetzt werden sollte, dass die Transformationssprache (und die Realisierung in Form einer Transformationsmaschinerie) eine definierte und bekannte Semantik besitzt bzw. umsetzt.

4.2.2 Grundbegriffe und Konzepte

Es folgen nun Definitionen der wichtigsten GT-Konzepte und -Grundbegriffe, die im weiteren Verlauf der Arbeit immer wieder benötigt werden. Das Ziel soll hierbei allerdings nicht sein, eine vollständige Theorie zu entwickeln bzw. eine solche vollständig wiederzugeben. Vielmehr soll eine für das weitere Verständnis ausreichende Intuition der Begriffswelt erreicht werden.

Graph, Typgraph und Instanz-/Objektgraph

Graphen sind die Basis eines jeden GT-Ansatzes. Exemplarisch seien hier einfache gerichtete Graphen erwähnt.

Definition 4.1 (Graph [Ehr+06, S. 21]):

Ein Graph $G = (V, E, s, t)$ umfasst eine Menge V von Knoten (engl. Vertices), eine Menge E von Kanten (engl. Edges) sowie zwei (linkstotale) Funktionen $s, t : E \to V$, welche die Quelle (engl. Source) und das Ziel (engl. Target) einer jeden Kante definieren.

Die Darstellung von Kanten in Definition 4.1 mit Hilfe der beiden Funktionen s und t hat Vorteile gegenüber der „traditionellen“ (relationalen) Beschreibung von Kanten als Tupel der Art $E \subseteq V \times V$ (bzw., falls Mehrfachkanten erlaubt sind, $E \subseteq V \times L \times V$, mit L als Menge von Kantenmarkierungen). So unterscheidet sich die Darstellung der Kanten in Definition 4.1 nicht von derjenigen für Knoten; Knoten und Kanten sind folglich gleichwertige Entitäten. Auch ist es möglich, Graphen nach Definition 4.1 als

sog. *algebraische Struktur* aufzufassen [Ehr+06, S. 12] (was sich letztendlich auch im Namen des *Algebraischen Ansatz* widerspiegelt [And+99]). Dadurch gelten bewiesene, grundlegende Eigenschaften über algebraischer Strukturen sowie zu dem Umgang mit diesen automatisch auch für solche Graphen.

In der Praxis reichen solche einfache Graphen jedoch i. d. R. nicht aus, um Modelltransformation, beispielsweise auf Basis von *MOF*- bzw. *EMF*, hinreichend genau zu beschreiben. Instanzgraphen sollen im Falle entsprechender (Meta-)Modelle auch *Typinformationen* und *Attribute* beinhalten. Innerhalb des Algebraischen Ansatzes werden deswegen erweiterte Graphmodelle[12] verwendet, die eine Typisierung sowie Attribute berücksichtigen, vgl. [EPT04] und [Ehr+06, Kap. 8]. In [BET12] wird insbesondere eine Formalisierung mit Berücksichtigung wesentlicher *EMF*-Spezifika (zyklenfreie Containment-Beziehungen, bidirektionale Referenzen, Vererbung, etc.) sowie angepasster *GT*-Regeln mit sogenannten *Multi-* oder *Mengenelementen* (unter Rückgriff auf *amalgamierte* Graphtransformationen) vorgestellt.

Im Folgenden nutzen wir den Begriff *Typgraph* synonym zum bereits definierten Metamodellbegriff *Metamodell*, wie in der nachfolgenden Definition 4.2 festgehalten.

Definition 4.2 (Typgraph (vgl. [Ehr+06, Def. 2.6])):

Der Typgraph *definiert eine endliche, feste Menge von* Knotentypen*, über denen die Knoten der zugehörigen Modellgraphen getypt sind, sowie eine endliche, feste Menge von* Kantentypen *über denen die Kanten der Modellgraphen getypt sind. Dabei ist der Typgraph selbst wieder ein Graph.*

Die Typisierung kann im Falle des algebraischen Ansatzes als Typmorphismus zwischen Instanz- respektive Objektgraphen und Typgraph beschrieben werden. Wir vereinbaren hier, dass Typgraph *im Folgenden ein Synonym für den* Metamodell*-Begriff aus Definition 2.2 ist und legen folglich die EMF-Interpretation*[a] *zugrunde.*

[a] Für eine exakte Formalisierung sei auf [BET12; BET08] verwiesen.

Die mathematischen Details unterschiedlicher Graphmodelle sind auch durch die Festlegung auf *EMF* von untergeordnetem Interesse. Wenn im Folgenden von Objekt- respektive Modellgraphen, kurz Graphen, die Rede ist, dann sind *Objektdiagramme*, genauer *EMF*-Modelle nach Definition 2.1, gemeint. Dies ist in Definition 4.3 noch mal festgehalten.

Definition 4.3 (Modell-, Instanz-, Objektgraph):

Ein Modellgraph *bzw.* Objektgraph *ist ein (im zeitlichen Verlauf veränderliches)* (*EMF*-)Modell *nach Definition 2.1, das über einem Typgraphen nach Definition 4.2 getypt ist.*

Auf technischer Ebene ergeben sich durch die *EMF*-Abbildung auf Java damit folgende Korrespondenzen: (i) Knoten haben ihre Entsprechung in Java-Laufzeitobjekten, (ii) als Knotentypen dienen Klassen im *EMF*-Modell (also `EClass`-Instanzen und ggf. deren Repräsentationen als Java-Klassen, welche wiederum auch die Java-Typen der zuvor

[12] Beispielsweise wird in [Ehr+06, Teil 3] eine entsprechende Kategorie attributierter Graphen eingeführt, die sog. $\text{AGraphs}_{\text{ATG}}$.

erwähnten Java-Laufzeitobjekten darstellen), (iii) Kanten entsprechen Links und werden als Nicht-Void-Werte einfacher Java-Referenzen (für Assoziationen im *EMF*-Modell mit oberer Multiplizität von '1') oder Einträgen in `EList`[13]-Instanzen (für Assoziationen im *EMF*-Modell mit obere Multiplizität ungleich '1') verwaltet, und (iv) Kantentypen entsprechen Assoziationen im Metamodell, also *EMF*-Referenzen (genauer Instanzen von `EReference`).

Muster, Match und GT-Regel

Im Folgenden wird eine angepasste Definition für *GT*-Regeln, analog zu Definition 3.3, gegeben. In einem ersten Schritt wird dazu in Definition 4.4 der Begriff des Graphmusters eingeführt.

Definition 4.4 (Graphmuster):

Ein Graphmuster *p ist eine Modellschablone in Gestalt einer graphartigen Struktur, die einen (potentiell erlaubten) Teilausschnitt eines Modells beschreibt.*

Bei den Entitäten, aus denen das Graphmuster zusammengesetzt ist, handelt es sich um Object-Variables (OVs) und Link-Variables (LVs). Beide Arten von Entitäten beziehen sich ebenfalls jeweils auf einen Typ des Typgraphen/Metamodells aus Definition 4.3, sind also, in gewisser Weise, ebenfalls getypt.

Bei den Elementen des Musters handelt es sich jedoch *nicht* um Elemente des Modells selbst. Vielmehr lassen sie sich als *Variablen* vom jeweiligen Typ auffassen, die an tatsächliche Elemente des Modells gebunden werden können. Dies geschieht teilweise durch den Prozess der Mustersuche, bei dem sie an passende Knoten und Kanten des Modellgraphs gebunden werden,[14] teilweise durch externe Vorgaben.

Ausgehend von den Definitionen 4.4 und 3.3 werden *GT*-Regeln nun wie folgt definiert:

Definition 4.5 (*GT*-Regel):

Eine Graphtransformationsregel *r ist ein Paar von Graphmustern, die analog zu Definition 3.3 als LHS, L, und RHS, R, der Regel bezeichnet werden. Auch hier gibt es ein Interface K, das die Elemente enthält, die sich L und R teilen. Elemente des Graphmusters, die nur in $L \setminus K$ existieren, stehen für zu löschende Modellelemente. Elemente, die nur in $R \setminus K$ existieren, stehen für neu anzulegende Modellelemente. Die übrigen Elemente (alle in K) repräsentieren zu konservierende Kontextelemente im Modell, die ebenfalls für einen Match, vgl. Definition 4.6, vorhanden sein müssen.*

In Definition 3.4 wurde bereits das Konzept der Operationalisierung einer Regel eingeführt. Dabei wurde auch das Pattern Matching als Teil der Regelauswertung erwähnt. Für Graphmuster ist die Suche nach einer passenden Anwendungsstelle, dem Treffer bzw. Match, vgl. Definition 4.6, ein wesentlicher Schritt der Ausführung.

[13] Hiermit ist *EMF*-spezifische Implementierung des Java-Interfaces `java.util.List` gemeint.
[14] Im Algebraischen Ansatz lässt sich diese Abbildung durch *strukturerhaltende Morphismen* beschreiben.

Definition 4.6 (Treffer bzw. Match einer *GT*-Regel):

Als Treffer *(engl.* Match*) m eines Graphmusters bezeichnen wir einen Teilgraphen des Modellgraphen, auf den das Graphmuster durch eine strukturerhaltenden Abbildung vollständig abgebildet werden kann, ohne dass dabei eventuell vorhandene Ausschlussbedingungen, die sogenannten NACs, verletzt werden. Außerdem müssen alle (optional) vorhandenen Nebenbedingungen, wie Vorgaben für Attributwerte, für die Elemente des Teilgraphen erfüllt sein.*

Anwendbarkeit von Regeln und Anwendung

Nach der Auswahl einer *GT*-Regel zur Auswertung, ist die Anwendbarkeit der Regel nach Definition 4.7 zu prüfen. Grundlage hierfür ist die Bestimmung eins Treffers.

Definition 4.7 (Regelanwendbarkeit):

Eine Regel ist anwendbar*, wenn alle ihre Vorbedingungen zum Zeitpunkt der Regelauswertung erfüllt sind. Die Anwendbarkeit wird mittels der Mustersuche überprüft (dies beinhaltet auch die Überprüfung von Attributbedingungen und von NACs). Anwendbarkeit setzt voraus, dass mindestens ein gültiger Treffer im Modell existiert.*

Einen einzelnen erfolgreichen Ersetzungsschritt, der die erfolgreiche Auswertung einer einzelnen Regel auf einem Modell umfasst, bezeichnen wir im Folgenden als *Regelanwendung* jener Regel. In Definition 4.8 sind die dazu notwendigen Schritte zusammengefasst.

Definition 4.8 (Regelanwendung):

Als Regelanwendung *einer GT-Regel wird der komplette Vorgang bezeichnet, der sich aus den folgenden Teilschritten ergibt:*

(i) Überprüfung der Regelanwendbarkeit mittels Mustersuche inkl. der Überprüfung aller Bedingungen,
(ii) Auswahl eines Treffers im Modell als Anwendungsstelle der Regel,
(iii) Löschen der explizit und implizit zu entfernenden Elemente,
(iv) Anlegen der zu erzeugenden Elemente sowie Verbinden mit dem Kontext des Treffers,
(v) Durchführen aller notwendigen (optionalen) Attributmanipulationen.

Graphersetzungsschritt, Graphtransformation

Die Schritte (iii) - (v) der Regelanwendung, s. Definition 4.8, stellen den eigentlichen *Graphersetzungsschritt* dar. In der nachfolgenden Definition 4.9 wird der Begriff entsprechend definiert.

Definition 4.9 (Graphersetzungsschritt (vgl. z. B. [Gie+12, Def. 11])):

Ein konkreter Graphersetzungsschritt*, engl. Graph Rewriting Step, entspricht der isolierten Anwendung einer GT-Regel, r, auf Basis eines bestimmten Matches, m,*

für einen gegebenen Modellgraphen, G.
Ein Graphersetzungsschritt wird im weiteren Verlauf wie folgt notiert: $G \overset{r,m}{\Longrightarrow} G'$ (die Angaben von r und m können ggf. entfallen).

Nach der Einführung einzelner Graphersetzungsschritte, lässt sich darauf aufbauend das Konzept einer *Graphtransformation* als Sequenz solcher Schritte wir folgt definieren:

Definition 4.10 (Graphtransformation (s. [Gie+12, Def. 12])):
Eine Graphtransformation *ist eine nichtleere Sequenz von n Graphersetzungsschritten: $G_{start} \Rightarrow \cdots \Rightarrow G_n$*

4.2.3 Quellen für Nichtdeterminismus

Bei Graphtransformationen nach Definition 4.10 existieren grundsätzlich zwei Quellen für Nichtdeterminismus. Zum einen ist bislang ungeklärt, wie einzelne Regeln für eine Anwendung ausgewählt werden. Selbst wenn nur tatsächlich solche Regeln berücksichtigt würden, deren jeweilige Anwendbarkeit gegeben wäre, könnte es i. Allg. dennoch vorkommen, dass mehrere Regeln miteinander konkurrieren. Andererseits ist bisher auch nicht festgelegt, wie die Auswahl eines Treffers (von potentiell vielen möglichen Treffern) für den Musterteil einer bestimmten Regel konkret erfolgt. Grundsätzlich sind alle validen Treffer aus Sicht der Regel gleich „gut“.

Regelauswahl

Bei den meisten GT-Formalisierungen, insbesondere dem Algebraischen Ansatz, stehen alle Regeln der Regelmenge gleichberechtigt auf einer Stufe nebeneinander. Es existiert also keine natürliche Sortierung, die festlegt, welche der anwendbaren Regeln zu einem Zeitpunkt[15] tatsächlich auch ausgeführt wird bzw. werden. Grundsätzlich gilt hierbei die Annahme, dass bei der Auswahl und Anwendung von Regeln von einer schrittweisen (bzw. „operationellen“) und „synchronen“ Ausführung (sollten mehrere Regeln angewendet werden, so erfolgt dies frei von Interaktion und *parallel* innerhalb eines Schrittes) ausgegangen wird. Für eine Beschreibung der Konzepte *parallele* und *sequenzielle Unabhängigkeit* von Regeln sei, für das Beispiel einer algorithmischen Formalisierung, auf [Roz97, Kap. 3.2.2, ab S. 172] verwiesen.

Da sich der Nichtdeterminismus bzgl. der Regelauswahl von Fall zu Fall mal als *günstig* ((Unter-)Spezifikation, Abstraktion, Nachweis bestimmter Eigenschaften unter Vernachlässigung konkreter Regelauswertungssequenzen), mal als *ungünstig* (z. B. im Hinblick auf Eigenschaften konkreter Implementierungen, dem Nachweis, dass eine Transformation deterministisch ist oder der Wartbarkeit) erweisen kann, wurden für den negativen Fall verschiedene Konzepte eingeführt, um die Regelauswahl zu kontrollieren. Wie in [Sch97, S. 506] ausgeführt, existieren hierfür mehrere Möglichkeiten, wie (i) Regelprioritäten, (ii) Reguläre Ausdrücke oder anderweitige (textuelle) Sprachen (z. B. *Abstract State Machines (ASMs)*,[16]) zur Beschreibung zulässige Regelsequenzen oder (iii) (visuel-

[15] Dies bezieht sich auf eine diskrete Interpretation des Zeitbegriffs.
[16] *ASMs* werden beispielsweise bei der VIATRA-Sprache [VB07; Cse+02] eingesetzt.

le) Kontrollflussgraphen. (Die Möglichkeit, konkrete Regeln so zu formulieren, dass ihre Auswertung aufgrund inhärenter Abhängigkeiten nur in einer bestimmten Reihenfolgen möglich ist, bleibt davon unberührt.) Ansätze, die den beiden letztgenannten Optionen (ii) und (iii) entsprechen, werden auch als *programmierte Graphersetzungssysteme* bezeichnet [Sch97] und haben ihren Ursprung in formalen und programmierten (Graph-) Grammatiken (vgl. hierzu [Bun79]). Das zuvor bereits erwähnte Werkzeug *PROGRES* ist ein wichtiger Vertreter früher Graphersetzungssysteme mit *programmierter* Regelanwendung [Sch91b; Sch91a; Zun98].

Im weiteren Verlauf der Arbeit sind nur noch solche programmierten Graphtransformationen Betrachtungsgegenstand. Die im kommenden Abschnitt 4.3 beschriebene Graphtransformationssprache nutzt beispielsweise Kontrollflussgraphen zur Steuerung der Regelanwendung. Falls vorausgesetzt wird, dass nichtdeterministische Verzweigungen unzulässig sind, führt dies dazu, dass die hier beschriebene Art des Nichtdeterminismus ausgeschlossen ist. Diese Einschränkung muss i. Allg. nicht gelten; für den weiteren Verlauf der Arbeit ist diese Einschränkung allerdings als gegeben anzunehmen.

Trefferauswahl

Die zweite Quelle für Nichtdeterminismus besteht darin, dass für eine anzuwendende Regel mehrere Anwendungsstellen im Graphen existieren können. Ob die potentiellen Anwendungsstellen voneinander unabhängig sind oder ob die Anwendung an einer der möglichen Stellen Auswirkungen auf die Anwendbarkeit an der oder den anderen Stelle(n) hat, ist hierbei unerheblich. Falls mehrere Regelanwendungsoptionen existieren, gibt es verschiedene Strategien, um mit dieser Situation umzugehen. Eine erste (deterministische) Option wäre es, *alle* konfliktfreien Anwendungsoptionen gleichzeitig zu nutzen, und die Regel für n Anwendungsstellen auch n-mal anzuwenden. Bei kollidierenden Anwendungsoptionen könnte der Zustand vor der Regelauswertung geklont werden, um im Anschluss jede Anwendungsstelle einzeln und separat zu nutzen. Eine dritte Option besteht darin, den Benutzer auswählen zu lassen. Die typische Option besteht dagegen darin, *eine* der möglichen Anwendungsstellen *nichtdeterministisch* auszuwählen, und die Regel für diesen Treffer auszuwerten. Dabei kann die tatsächliche Auswahl auf technischer Ebene, aufgrund der Art der Implementierung dennoch deterministisch erfolgen oder aber sich komplett der Vorhersagbarkeit entziehen, beispielsweise bei (echt) zufälliger Auswahl.

Wird diese nichtdeterministische Wahl der Anwendungsstelle von Regeln mit der (deterministischen) Auswahl der anzuwendenden Regel per Kontrollflussgraphen kombiniert – dies ist beispielsweise bei der im nächsten Abschnitt vorgestellten Transformationssprache der Fall – so hat dies interessante Konsequenzen: Es kann vorkommen, dass bei mehrfacher Ausführung einer Transformation (mit genügend komplexem Kontrollfluss und entsprechend vielen Regeln) auf jeweils dem gleichen (und ausreichend großen) Eingabegraphen, unterschiedliche Pfade durch den Kontrollflussgraphen genommen werden. Mitunter unterscheiden sich hierdurch die Ausgaben *grundlegend* voneinander, da beispielsweise durch ungünstige Wahl einer Anwendungsstelle keine Anwendungsstellen mehr für eine nachgelagerte Regel existieren, also der Prozess in eine (evtl. vermeidbare) Sackgasse läuft. Bei der Verwendung entsprechender Sprachen trägt der Entwickler die alleinige Verantwortung dafür, sicherzustellen, dass die Transformation trotzdem noch korrekte Ergebnisse (im Sinne der Anforderungen) liefert. So bleibt festzuhalten, dass dieser Aspekt eine potentielle Fehlerquelle bei der Umsetzung von *GTs* sein kann.

Eine weitere andere Art, um mit dieser Art des Nichtdeterminismus umzugehen, wurde z. B. in PROGRES umgesetzt, vgl. [Fis+00]. Sollte die Anwendung einer Regel a dazu führen, dass die Anwendung einer nachfolgenden Regel b ausgeschlossen ist, wird unter Rücknahme der Auswertung von a per *Backtracking* so lange eine andere Anwendungsstelle für a gesucht, bis entweder b anwendbar ist oder alle Optionen vergeblich getestet wurden. (In letzterem Fall wird eine vordefinierte *Ausnahme* ausgelöst.) Allerdings kann auch diese Strategie nicht verhindern, dass bei identischer Eingabe unterschiedliche Ergebnisse möglich sind. Es wird nur sichergestellt, dass die GT nicht verfrüht zum Erliegen kommt. Die *Konfluenz* der GT – also die Eigenschaft der GT, dass für eine bestimmte Eingabe bei allen möglichen Ableitungssequenzen ein äquivalentes Ergebnis entsteht – muss separat gezeigt werden.

4.3 Die SDM-Sprache

Im Folgenden wird eine konkrete GT-Sprache vorgestellt, auf die sich der eigentliche Beitrag der vorliegenden Arbeit bezieht. Die Sprache wird aufgrund ihrer Entstehungsgeschichte und der Tatsache, dass sie ursprünglich als zentraler Bestandteil der als *Story Driven Modeling* bezeichneten Entwicklungsmethodik [ZSW99; NZJ13] entwickelt wurde, als *SDM-Sprache* oder kurz *SDM* bezeichnen. Dabei wird in dieser Arbeit bewusst die Verwendung der in der Literatur häufig vorkommenden Bezeichnung „*Story-Diagramme*“ bzw. *Story-Diagramm*-Sprache [Fis+00; Zun02] vermieden, um dadurch zu verdeutlichen, dass es sich um einen leicht angepassten und um einige Sprachmittel reduzierten Dialekt handelt.

Die *SDM*-Sprache ist das Resultat zweier Entwicklungen in jüngerer Vergangenheit, nämlich (i) der Freigabe und kontinuierlichen Weiterentwicklung des *eMoflon*-Werkzeugs[17,18] sowie (ii) der Entwicklung eines gemeinsamen, werkzeugübergreifenden *SDM*-Metamodells, vgl. hierzu auch die Abbildung E.1 bis E.7 aus Anhang E, im Rahmen der *sdm-commons*-Initiative.[19] Einige Bekanntheit und Verbreitung haben Story-Diagramme als Teil des *Fujaba*[20]-Werkzeugs [Fis+00; FNT98; NNZ00] erlangt. Darüber hinaus nutzen seit einiger Zeit auch Nachfolgeprojekte und deren Werkzeuge die Sprache für *MTs*. Insbesondere sind dies *Fujaba4Eclipse*[20] [Pri+10], *MDELab*[21] [GHS09] und *MOFLON* [Ame+06] respektive *eMoflon* [Anj+11].

Bei der *SDM*-Sprache handelt es sich um einen Ansatz für *programmierte GTs* und um eine Vertreterin des algorithmischen Ansatzes. Ursprünglich wurde sie für den Einsatz im *CASE*-Umfeld entwickelt. Sie weist einen starken Bezug sowohl zu *UML* (konkrete Syntax) als auch Java (als Ziel- und Hostsprache sowie Schnittstelle zu externer Funktionalität) auf. Als ursprüngliche Hauptgründe für die Entwicklung von Story-Diagrammen wurden (i) der proprietäre Charakter bestehender Ansätze sowie (ii) die fehlende bzw. schlechte Integration in den *OO*-Kontext in [Fis+00] genannt. Fujaba kann als De-facto-Nachfolger von PROGRES angesehen werden und ist mit diesem nicht nur in personeller

[17] Nachfolger von MOFLON, vgl. [Ame+06], s. auch Fußnote 18
[18] `http://www.emoflon.de/`
[19] `https://code.google.com/a/eclipselabs.org/p/sdm-commons/` (zuletzt abgerufen am 4.7.2014)
[20] From Uml to Java And Back Again, vgl. auch `http://www.fujaba.de/` (zuletzt abgerufen am 1.7.2014)
[21] `http://www.mdelab.de/` (zuletzt abgerufen am 1.7.2014)

Hinsicht – Prof. A. Zündorf hat sowohl im PROGRES-Umfeld promoviert als auch die Fujaba-Entwicklung federführend initiiert – sondern auch auf konzeptioneller Ebene eng verbunden.

Im Allgemeinen dienen Story- bzw. *SDM*-Diagramme der Beschreibung bzw. der Implementierung des Verhaltens der Operationen von Klassen im Metamodell. Dies geschieht vornehmlich in Form sogenannter *Story-Patterns* – visueller *GT*-Regelbeschreibungen, bestehend aus einer kombinierten Darstellung von *LHS* und *RHS*, bei der die zu erhaltende Elemente des Interfaces durch schwarze, die zu löschenden Elemente der *LHS* durch rote und die zu erzeugende Elemente der *RHS* durch grüne Knoten und Kanten angegeben werden. Die konkrete Syntax von Knoten und Kanten einer *GT*-Regel ist – abgesehen von der farblichen Kennzeichnung der Elemente – an die Syntax von Objekt- bzw. Kommunikationsdiagramme der *UML* angelehnt.

Die *GT*-Regeln sind eingebettet in einen (ebenfalls grafisch spezifizierten) *Kontrollflussgraphen*. Dessen Darstellung orientiert sich wiederum an der Syntax einfacher *UML*-Aktivitätsdiagramme. Einzelne *Aktionen* (nach Aktivitätsdiagrammnomenklatur) können eine *GT*-Regel beinhalten, und werden dann als *Story*-Knoten respektive *Story-Node* bezeichnet. Ein einzelnes Aktivitätsdiagramm repräsentiert eine in sich geschlossene funktionale Einheit und ist genau einer Operation (einer Klasse des Metamodells) zugeordnet.

Die Semantik des Graphtransformationsteil von *SDM*-Beschreibungen lässt sich (näherungsweise) mit Hilfe der *SPO*-Interpretation des algebraischen Ansatzes beschreiben, wie z. B. in [Gie+12; GZ06] vorgeschlagen. So werden beispielsweise auch die in den Regeln nicht aufgeführten Kanten, die an zu löschenden Knoten anhaften, als Seiteneffekte der Regelanwendungen mit entfernt. Allerdings besitzen die ursprünglichen Story-Diagramme, wie sie in [Fis+00; Zun02] beschrieben sind, auch Konstrukte (z. B. sog. *Mengenknoten*[22]) deren Semantik sich nicht mehr mit Hilfe dieser ursprünglichen Standardtheorie fassen lässt.[23] Die ursprüngliche Semantikdefinition erfolgte auch nicht durch einen algebraischen sondern durch einen algorithmischen Ansatz [Zun02, Abschnitt A.3] – konkret durch eine Abbildung auf die Semantik von PROGRES [Fis+00].

Ausführbar werden Story-Diagramme entweder durch Abbildung auf (Java-)Code mit Hilfe eines Codegenerators [Zun02; GSR05; GBD07], wie bei Fujaba, MOFLON und eMoflon, oder aber durch den Einsatz eines Interpreters, wie bei MDELab [GHS09]. Der Vorteil eines generativen Ansatzes liegt vornehmlich in einer höheren Laufzeitperformanz sowie einer besseren Portierbarkeit durch den Verzicht auf einen gesondert zu portierenden Interpreter. Insbesondere bei Fujaba besteht auch die grundsätzliche Möglichkeit zur Generierung von Code anderer Zielsprachen oder der Nutzung alternativer *APIs*. Der interpretierte Ansatz bietet dagegen Möglichkeiten zur Rückwärtsausführung und/oder verbesserte Debugging-Funktionen. Beide Ausführungsarten legen letztendlich die *tatsächliche, implementierte* Semantik der Sprache fest. Diese muss, im Falle von Inkonsistenzen und Fehlern, nicht notwendigerweise mit der ursprünglich gedachten, for-

[22] Mengenknoten sind eine Schreibabkürzung für Muster beliebiger Größe. Sie bedingen eine Art des Greedy-Verhaltens bei der Mustersuche, welches konzeptuelle Ähnlichkeiten zum Verhalten der "*"- und "."-Operatoren bei regulären Ausdrücken aufweist. Ein Treffer wird hierdurch um maximal viele Objekte eines bestimmten Typs erweitert.

[23] Es existieren Erweiterungen der algebraischen Theorie, bei denen durch wiederholte *Amalgamierung* von Regeln und Regelteilen Mengenknoten bzw. vergleichbare Konzepte formalisierbar werden, vgl. z. B. [Bie+10a].

malen Semantik übereinstimmen, weshalb ggf. auch die Korrektheit der Codegenerierung bzw. der Interpretierung ggf. gesondert überprüft werden muss.

4.3.1 Kontrollfluss

Wie bereits erwähnt, wird bei *SDM*-Transformationen die Regelanwendung durch einen Kontrollflussgraphen gesteuert, welcher visuell modelliert wird. Die Notation orientiert sich an *UML*-Aktivitätsdiagramme ohne deren komplexe Konstrukte und Konzepte wie Objektflüsse, Parallelität (Teilungs- und Synchronisationsknoten), Parameter, Partitionen, Datastores oder Signale.

In Abbildung 4.1 ist ein minimales Beispiel einer *SDM*-Transformation dargestellt. Für die Klasse `AClassWithBehavior`, für die eine parameterlose Operation `doSomething` mit Rückgabetyp `EBoolean` definiert ist, ist die rudimentäre Implementierung der Operation im rechten Bereich der Abbildung abgebildet. Der Story-Knoten `ActivityNode1` ist hier leer, um nicht von dem Kontrollfluss abzulenken.

Bereits dieses einfache Beispiel zeigt die wesentliche Struktur eines Story- bzw. *SDM*-Diagramms. Der *Startknoten* ist mit der Signatur der Operation und dem Klassennamen annotiert und definiert den Einstiegspunkt. Mit Hilfe von *Transitionen* und weiteren *Knoten* (hier: ein *Story-Knoten* und ein *Stoppknoten*) wird der Kontrollflussgraph einer Operation gebildet.

Stoppknoten definieren das Ende eines Ablaufs. Sie können einem Ausdruck zur Angabe eines Rückgabewertes umfassen – für das konkrete Beispiel beschreibt der Stoppknoten, dass bei dem Verlassen der Operation stets das Literal `false` zurückgeliefert wird. Die eigentlichen *GT*-Regeln sind Inhalt einzelner Story-Knoten und werden ausgewertet, sobald der Kontrollflussablauf den entsprechenden Knoten erreicht. In diesem Beispiel nicht zu sehen, aber grundsätzlich möglich: der Kontrollfluss kann mit Hilfe von Fallunterscheidungen verzweigt werden. Die Möglichkeit zur Verzweigung besteht grundsätzlich bei allen Story-Knoten (sowie bei sog. Statement-Knoten).

Nach dieser ersten Übersicht zum Kontrollfluss, betrachten wir im Folgenden die Elemente aus denen der Kontrollflussgraph aufgebaut ist im Detail. Im Anschluss daran wird kurz die Wohlgeformtheit von Kontrollflussgraphen und speziell von Story-Diagrammen thematisiert.

Sprachmittel des SDM-Kontrollflusses

In Abbildung E.2, Anhang E, ist der Teil des *SDM*-Metamodells dargestellt, der den Kontrollflussgraphen von *SDM*-Diagrammen beschreibt. Basisklasse der *EMF*-Containment-Hierarchie und Repräsentation der Instanzen kompletter *SDM*-Diagramme ist die Klasse `Activity`. Sie referenziert einerseits die `EOperation`, deren Verhalten durch das *SDM*-Diagramm definiert wird. Zusätzlich enthält eine `Activity`-Instanz alle *(Kontrollfluss-)Knoten* (Instanzen konkreter Unterklassen von `ActivityNode`) und *(Kontrollfluss-)Kanten* (`ActivityEdge`) der *SDM*. Die Bezeichner *Activity*, *ActivityNode* und *ActivityEdge* gehen auf den *UML*-Metamodellteil zur Beschreibung von Aktivitätsdiagramme zurück, vgl. z. B. [11c, Abbildungen 12.2 und 12.5].

Die Kanten des Kontrollflussgraphen werden, unter anderem in Anlehnung an die ältere UML 1.x-Nomenklatur, auch als *Transitionen* bezeichnet. Sie verknüpfen einen einzelnen Quellknoten (`source`-Referenz) mit genau einem Zielknoten (`target`-Referenz)

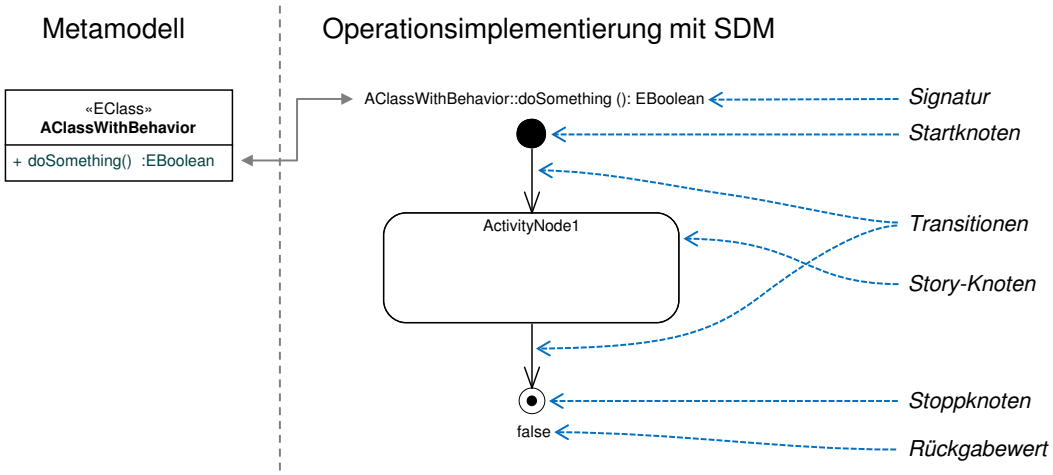


Abbildung 4.1: Die wichtigsten Notationselement bei SDM-Diagrammen

und stehen für einen möglichen Übergang des virtuellen „Kontrollflusstokens“ entlang der Pfeilrichtung. Bezogen auf die Zulässigkeit von Transitionen bestehen einige mehr oder weniger offensichtliche Einschränkungen. Beispielsweise sollte unmittelbar einzusehen sein, dass Startknoten nicht Ziel und Stoppknoten nicht Quelle einer Transition sein können. Ferner tragen die Transitionen eines von fünf vordefinierten Wächtersymbolen (engl. Guards), deren jeweilige Bedeutungen im weiteren Verlauf erläutert werden. Standardmäßig ist das `guard`-Property mit `EdgeGuard.NONE` belegt, was bedeutet, dass eine entsprechende Transition „ohne Einschränkung“ ist und es neben ihr keine weiteren parallelen Transitionen geben darf. Egal, wie das Ergebnis der Auswertung des Quellknotens ausfällt, der weitere Ablauf erfolgt entlang einer solchen Transition. Die Darstellung einer Transition mit diesem Guard-Typ lässt sich aus Abbildung 4.1 entnehmen.

Startknoten Startknoten werden im *SDM*-Metamodell durch die `StartNode`-Klasse modelliert, s. Abbildung E.2. In Funktion und visueller Darstellung entsprechend sie weitestgehend dem *InitialNode* von *UML*-Aktivitätsdiagrammen. Ein valides *SDM*-Diagramm umfasst *genau einen* Startknoten mit genau einer ausgehenden Transition (ohne Guard). Eingehende Transitionen bei einem Startknoten wurden bereits als unzulässig ausgeschlossen. In der konkreten Syntax umfasst der Startknoten eine textuelle Repräsentation der (im beinhaltenden Diagramm) implementierten Operation.

Stoppknoten Stoppknoten, vgl. die `StopNode`-Klasse im Metamodell, Abbildung E.2, entsprechen in Funktion und Aussehen den *ActivityFinal*-Knoten von Aktivitätsdiagrammen. In einem validen *SDM*-Diagramm muss mindestens ein Stoppknoten vorhanden sein und jeder bei einem Startknoten entspringender Ablauf muss bei einem Stoppknoten enden. Folglich hat ein Stoppknoten in einem validen Diagramm mindestens eine eingehende und keine ausgehenden Kanten. Um den Rückgabewert einer Operation zu spezifizieren, umfasst ein Stoppknoten optional einen Ausdruck zur Festlegung des Wertes, vgl. die Containment-Kante zwischen `StopNode` und `Expression` in Abbildung E.2. Die Frage, welche konkreten Ausdrücke möglich sind, wird später beantwortet.

Story-Knoten Betrachten wir nun die `ActivityNode`-Unterklassen, deren Bedeutungen sich weniger offensichtlich erschließen. Von den beiden noch unerwähnten Kontrollflussknoten sind die sog. *Story-Knoten*, vgl. hierzu die Klasse `StoryNode` im Metamodell, Abbildung E.2), die häufiger genutzten. Sie dienen der Einbettung einer einzelnen *GT*-Regel in den Kontrollflussgraphen und ermöglichen so deren Auswertung. In der konkreten Syntax werden sie analog zu *UML-Actions* durch Rechtecke mit abgerundeten Ecken dargestellt. Die eigentliche *GT*-Regel wird in Gestalt eines sog. Story-Musters berücksichtigt, welches im *SDM*-Metamodell durch die Klasse `StoryPattern` modelliert ist. Da grundsätzlich zwei grundlegende Ergebnisse bei der Regelauswertung möglich sind – entweder kann für den Musterteil der Regel ein Treffer im Modell gefunden werden (und einer der Treffer wird durch Anwenden der Regel erfolgreich umgeschrieben) oder eben nicht – stellt jeder Story-Knoten einen Entscheidungspunkt einer entsprechenden Fallunterscheidung dar. Anhand dieser Entscheidung lässt sich der Kontrollfluss verzweigen, so dass beispielsweise beide Fälle gesondert behandelt werden können.

Folglich kann ein *einfacher*[24] Story-Knoten genau (i) eine (für den Fall, dass keine Fallunterscheidung anhand der Regelauswertung gewünscht ist) oder (ii) zwei (für den Fall einer Fallunterscheidung) *ausgehende* Kontrollflusskanten aufweisen. Wichtig ist, dass im Falle (i) die ausgehende Kante keinen Guard (in der konkreten Syntax) tragen darf, im Falle (ii) dagegen zwei spezifische Guards vorgeschrieben sind: Der `SUCCESS`-Guard markiert den Verlauf des Kontrollflusses für den Fall der erfolgreichen Treffersuche (und Regelauswertung) und ist im Diagramm durch die textuelle Annotation „[Success]“ gekennzeichnet. Der `FAILURE`-Guard, welcher durch die Annotation „[Failure]“ ausgezeichnet ist, repräsentiert dagegen der Kontrollflusszweig, der genommen wird, falls kein gültiger Treffer für das Muster gefunden werden konnte. In Abbildung A.7 lassen sich diese beiden Fälle am Beispiel des Story-Knotens `split_the_add_block` nachvollziehen.

For-Each-Knoten Neben den oben beschriebenen *einfachen* Story-Knoten gibt es mit dem *For-Each*-Knoten einen weiteren Story-Knoten mit angepasster Semantik. Im Metamodell ist dieser Knotentyp nicht in Form einer dedizierten Klasse berücksichtigt. Vielmehr handelt es sich auch um `StoryNode`s, bei denen allerdings das `forEach`-Attribut, vgl. Abbildung E.2, den Wert `true` trägt. In der Visualisierung unterscheiden sich For-Each-Knoten von einfachen Story-Knoten durch die angedeutete „Stapelung“[25] zweier leicht versetzter Story-Knoten, wie sie beispielsweise in Abbildung A.4 mehrfach zu erkennen ist.

Der semantische Unterschied bezieht sich auf eine veränderte Art und Weise der *GT*-Regelauswertung, insbesondere hinsichtlich der Anzahl der Auswertungen. Zusätzlich ergeben sich auch Unterschiede in Bezug auf die Einbettung in den Kontrollfluss. So wird die Regel im For-Each-Fall *wiederholt* und für *jeden möglichen Treffer im Modell* ausgewertet, statt einmalig für einen (nicht-deterministisch) ausgewählten, wie im Falle einfacher Story-Knoten. Auch lässt sich für komplexe Bearbeitungsabläufe auf Basis eines einzelnen solchen Treffers, die sich nicht mehr mit einer einzelnen Regel beschreiben lassen, ein Teilablauf im Kontrollflussgraphen spezifizieren, der für die individuelle Wei-

[24] Die Bedeutung des Adjektivs „einfach“ erschließt sich bei der Beschreibung des anschließenden Konzepts.

[25] Diese Art der Darstellung ist vergleichbar zur Darstellung sogenannter Multi-Objekte in UML 1.x, vgl. [03b, S. „3-127“].

terverarbeitung jedes Treffers durchlaufen wird. Hierzu werden *einmalig* alle (konflikt- und überlappungsfreien) Treffer im Modell für das For-Each-Muster bestimmt und *parallel unabhängig* transformiert. Technisch erfolgt die Regelauswertungen aufgrund der hier genutzten Abbildungsart des Konstrukts auf den Zielsprachencode allerdings implizit sequenziell für die einzelnen Treffer. So wird die Auswertung für einen Treffer erst komplett durchlaufen bevor im Anschluss daran der nächste Treffer bearbeitet wird. Dieses Vorgehen simuliert die parallele Auswertung, so lange ausgeschlossen ist, dass bei der Auswahl des nächsten Treffers neue Treffer berücksichtigt werden, die sich grundsätzlich aufgrund der vorherigen Auswertung ergeben könnten. (Würden Treffer durch die vorherige Auswertung entfallen, wäre dies zumindest aus Terminierungssicht kein Problem.) Sollten bei der erneuten Auswertung der Regel neue Treffer (aufgrund vergangener Anwendung der Regel oder durch Seiteneffekte im Each-Time-Teil, s. u.) mit berücksichtigt werden, so würde das Verhalten dem einer Schleife gleichen und die Gefahr nichtterminierender Abläufe beinhalten.

Der entsprechend wiederholt zu durchlaufende Teilgraph des Kontrollflussgraphen ist an seinem For-Each-Knoten „verankert“. Dies bedeutet, dass der Teilgraph nur über diesen For-Each-Knoten mit dem restlichen Graphen verbunden ist. Der Teilgraph kann nur durch eine vom For-Each-Knoten ausgehende Transition mit dem `EACH_TIME`-Wächter – im Diagramm durch die Annotation „[Each Time]“ ausgezeichnet – betreten und nur durch (mindestens eine) in den gleichen For-Each-Knoten einlaufende Transitionen (die bei einem „letzten“ Knoten im Ablauf durch den Teilgraphen entspringt) verlassen werden. Im Folgenden werden solche Teilgraphen als *Each-Time-Komponente* bezeichnet, da sich durch das Entfernen der zuvor beschriebenen Kanten eine isolierte (schwache) Zusammenhangskomponente ergeben würde. Ein For-Each-Knoten wird nach der Behandlung aller Treffer für das zugehörige Muster durch eine Kontrollflusskante mit `END`-Guard – in der konkreten Syntax durch die Annotation „[End]“ markiert – verlassen.

In Abbildung 4.2 ist ein einfaches *SDM*-Diagramm skizziert, das sowohl eine Verzweigung des Kontrollflusses bei einfachen Story-Knoten (`initial_rule` und `process_each_match`) mittels `SUCCESS`- und `FAILURE`-Kante, als auch bei einem For-Each-Knoten (`repeated_rule_evaluation`) mittels `EACH_TIME`- und `END`-Kante aufweist. Bei letzterem würde für jeden Treffer des Musteranteils der *GT*-Regel des For-Each-Knotens `repeated_rule_evaluation` die Regel im `process_each_match` Story-Knotens ausgewertet und bei einem Nichtvorhandensein eines Treffers (`FAILURE`-Kante) die nächste Regel des `corner_case_rule`-Knotens ausgewertet. (Alle *GT*-Regeln sind hier der Übersichtlichkeit halber ausgespart).

Statement-Knoten Bei dem letzten noch ausstehende Knotentyp handelt es sich um die sog. *Statement*-Knoten. Sie werden im *SDM*-Metamodell durch die Klasse `StatementNode` repräsentiert, vgl. Abbildung E.2. Die Klasse referenziert einen Ausdruck (engl. Expression), der eine einzelne auszuwertende Anweisung (engl. Statement) definiert. Statement-Knoten stellen eine Möglichkeit dar, um externe Funktionalität einzubinden, beispielsweise solche, die sich nicht oder nur sehr schwer mit *SDM*-Diagrammen modellieren ließe oder die bereits anderweitig umgesetzt wurde. Auch stellen sie eine prominente Möglichkeit dar, um Code der Wirtssprache Java zu nutzen.

Mit Hilfe von Statement-Knoten können Signale versendet oder externe Aktionen gestartet werden. Andererseits sind Statement-Knoten auch eine Möglichkeit, um in einem

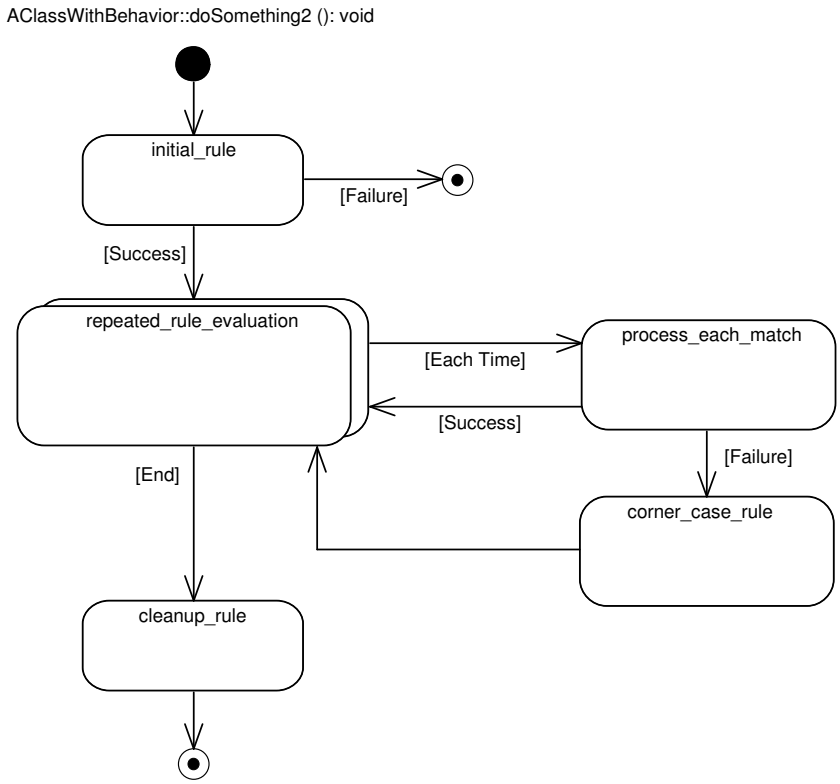


Abbildung 4.2: Ein SDM-Diagramm mit Kontrollflussverzweigung und For-Each-Knoten

SDM-Diagramm die Funktionalität anderer *SDM*-Diagramme zu nutzen, so dass (i) Teilberechnungen ausgelagert werden können, (ii) rekursive *SDM*-Invokationen ermöglicht werden oder aber (iii) der Kontrollfluss aufgrund eines Rückgabewertes einer Teilauswertungen verzweigt werden kann. Der erste Aspekt ist entscheidend für die Wiederverwendbarkeit und Modularisierung von *SDM*-Transformationen sowie eine funktionale Partitionierung. Der zweite Aspekt ermöglicht eine elegante Lösungsoption für viele Aufgaben und Algorithmen bei rekursiven Modellstrukturen (wie z. B. Bäumen), ohne dass dafür auf in *SDM* fehleranfällige Schleifen[26] zurückgegriffen werden muss. Rein auf Kontrollflussaspekte bezogen ist der dritte Punkt allerdings der wichtigste. Ausgehend vom Aufruf einer einzelnen Operation, die in einem der beteiligten Metamodelle eingeführt wurde und die einen Rückgabetyp besitzt, der entweder vom einfachen Ecore-Typ `EBoolean` oder von einem komplexen Typ im Metamodell ist, können die folgenden beiden Fälle unterschieden werden: Bei `EBoolean` kann die Rückgabe entweder `false` oder `true` lauten. Bei komplexen Typen dagegen sind die Möglichkeiten `null`, was `false` entspricht, bzw. ungleich `null`, was `true` entspricht. Demgemäß kann der Kontrollfluss mittels `FAILURE`- (Fall `false`/`null`) und `SUCCESS`-Kante (Fall `true`/$\neq$`null`) bei einem Statement-Knoten anhand des Rückgabewerts verzweigt werden.

[26] Damit sind keine For-Each-Abläufen gemeint, sondern For- und While-artige Schleifen im Kontrollfluss.

Wohlgeformtheit des Kontrollflussgraphen

Weiter oben wurden bereits einige Bedingungen formuliert, die ein valider Kontrollflussgraph zu erfüllen hat. Weitere Einschränkungen fanden bereits im Rahmen der Modellierung in das *SDM*-Metamodell Berücksichtigung. Darüber hinaus existieren weitere Einschränkungen. Alle sich ergebenden Einschränkungen werden unter dem Begriff der *Wohlgeformtheit* zusammengefasst.

SDM-Kontrollflussdiagramme sind Flussdiagrammen[27] (engl. Flowchart, Flow Diagram) nicht unähnlich. Die „graphartige“ Natur solcher Beschreibungen ermöglicht es grundsätzlich, einen *unstrukturierten*, Go-To-artigen Programmablauf zu spezifizieren. Diese Unstrukturiertheit, die sich mittels formaler Eigenschaften charakterisieren lässt, vgl. hierzu beispielsweise [All70; HU72; HU74], ist *keine* notwendige Voraussetzung, um alle *möglichen* Berechnungen durchzuführen zu können.[28] Und so wird die nichtstrukturierte Art der Programmierung gemeinhin als schlechter Programmierstil gesehen [Dij68].

Bereits für die ursprünglichen Story-Diagramme aus [Fis+00] wurde betont, dass sie wohlgeformt sein müssen, auch um die Semantik der Sprache durch eine Abbildung auf die PROGRES-Semantik definieren zu können. Das folgende Zitat belegt dies:

> „[...] Story Diagrams are restricted to so-called **well-formed control flow**. Basically, the control flow built by the transitions of a Story Diagram must represent nested sequences, branches, and loops. This enables a translation into Progres [PROGRES, Anm. des Autors] sequence-, choose-, and loop-statements. [...]“ [Fis+00, S. 302]

Der klassische Strukturiertheitsbegriff für Flowcharts greift allerdings zu kurz, da bei Story-Diagrammen die zuvor erwähnten Each-Time-Komponenten existieren. Diese müssen entsprechend gesondert berücksichtigt werden, z. B. indem man einzelne dieser Komponenten als eigenständige Diagramme interpretiert und zusätzliche Bedingungen für die notwendigen Verbindungen mit dem For-Each-Knoten formuliert. Unabhängig von den Details, gehen wir davon aus, dass ein (nach geeigneter Definition) strukturierter Kontrollfluss eine notwendige Bedingung für ein valides und korrekt funktionierendes Story-Diagramm darstellt. Für *SDM*-Diagrammen gilt diese Anforderung analog.

Zusammengefasst, existieren folgende Anforderungen:

- Bzgl. Startknoten soll gelten
  - genau ein Startknoten pro Diagramm
  - keine eingehenden Kanten
  - genau eine ausgehende Kante ohne Wächter
  - nicht mit einem Stoppknoten verbunden
- Bzgl. Stoppknoten soll gelten

[27] Auch bekannt unter der Bezeichnung *Programmablaufplan* (s. auch DIN 66001 und ISO 5807:1985).

[28] Auch strukturierte Diagramme, mit den drei Sprachkonzepten *sequenzielle Ausführung*, *Fallunterscheidungen* und *Schleifen*, sind grundsätzlich ausdrucksmächtig genug, wie in [BJ66] nachgewiesen. Darüber hinaus lassen sich Go-To-Programmen auch in eine Go-To-freie Darstellungen mit gleichen Endergebnissen und Topologien (ggf. unter Zuhilfenahme von zusätzlichen Variablen) algorithmisch ableiten [AM71]. Die verschiedenen Darstellungen sind allerdings i. Allg. nicht in allen Belangen äquivalent. Es können sich beispielsweise Unterschiede bzgl. der Berechnungsreihenfolge ergeben, vgl. [AM71; Kas74].

  - mindestens ein Stoppknoten pro Diagramm
  - mindestens eine eingehende Kante
  - keine ausgehenden Kanten
  - Rückgabewert entweder leer oder passend zu Rückgabetyp

- Für jeden Story-Knoten soll gelten
  - mindestens eine eingehende Kante
  - eine ausgehende Kante ohne Wächter oder zwei ausgehende Kanten mit den Wächtern `SUCCESS` und `FAILURE`

- Für jeden For-Each-Knoten soll gelten
  - mindestens eine eingehende Kante und, falls vorhanden, mindestens eine eingehende Kante aus der Each-Time-Komponente
  - eine ausgehende `END`-Kante, die nicht in die Each-Time-Komponente einläuft
  - optional: genau eine ausgehende `EACH_TIME`-Kante in die Each-Time-Komponente

- Für jeden Statement-Knoten soll gelten
  - mindestens eine eingehende Kante
  - bei Aufruf einer Operation mit dem Rückgabetyp `EBoolean` *oder* einer Klasse des Metamodells als Rückgabetypen sind zwei ausgehende Kanten möglich (`SUCCESS` und `FAILURE`); ansonsten genau eine ausgehende Transition ohne Wächter

- Keine vom Startknoten aus unerreichbaren Knoten oder Komponenten

- Von jedem Nicht-Stoppknoten muss ein Pfad zu einem Stoppknoten existieren

- Selbsttransitionen sind nicht erlaubt

Für Each-Time-Komponenten muss darüber hinaus gelten, dass alle Abläufe durch einen solchen Teilgraphen bei dem zugehörigen, steuernden For-Each-Knoten entspringen und auch dort wieder enden: Abläufe dieser Art entsprechen folglich einem *geschlossenen* Kantenzug durch den Graphen, mit der Einschränkung, dass Teile außerhalb der Komponente nicht von Knoten der Komponente aus durch Kantenfolgen erreichbar sind, die *nicht* zuvor den steuernden „Anker"-For-Each-Knoten (im Rahmen des zulässigen Verlassen des Teilablaufs) traversiert haben. Innerhalb einer Each-Time-Komponente sind weitere Each-Time-Knoten erlaubt (beliebig tiefe Verschachtelungen sind möglich). In Abbildung 4.3 sind valide und nichtzulässige Verbindungskanten skizziert.

4.3.2 Graphersetzung und Story-Patterns

Den Kern der *SDM*-Sprache bilden die Konstrukte zur Beschreibung der regelbasierten Graphtransformationen. *GT*-Regeln werden, wie bereits bei der Einführung der Story-Knoten erwähnt, in Form sog. *Story-Patterns*, vgl. die `StoryPattern`-Klasse im Metamodell (Abbildung E.6), angegeben. Die Beziehung zu einem Story-Knoten wird über eine entsprechende Assoziation hergestellt, vgl. Abbildung E.2. Die Darstellung von *GT*-Regeln in der konkrete Syntax, orientiert sich stark an der Darstellung von Objekt- bzw.

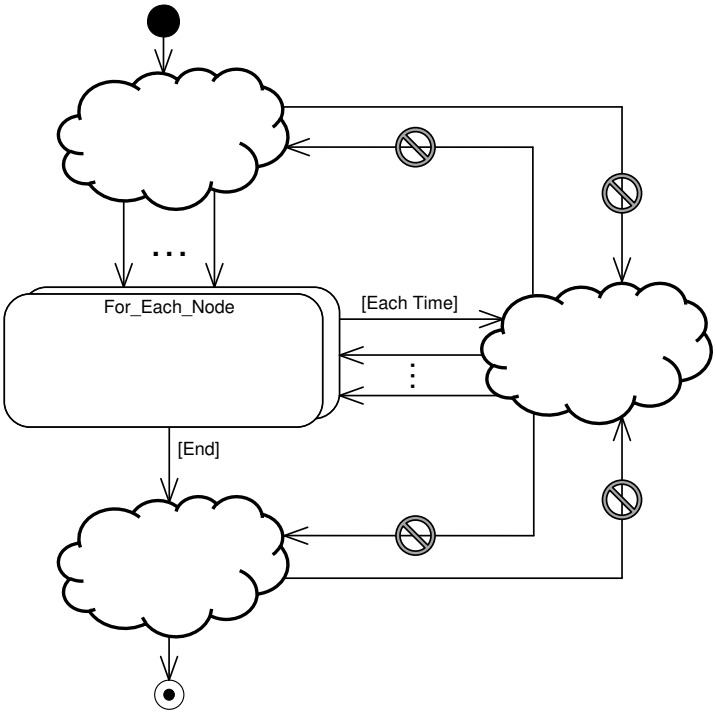


Abbildung 4.3: Valide und nichtvalide Kontrollflusskanten bei der Verwendung eines For-Each-Knotens mit Each-Time-Komponente (die Wolken symbolisieren valide Komponenten des Kontrollflussgraphen)

Kommunikationsdiagrammen der *UML*. Zusätzlich bestimmen textuelle Annotationen („++“ und „−−“) und/oder die Farbe von Elementen (Rot, Grün und Schwarz), ob ein Element Teil der linken oder der rechten Regelseite bzw. des Regel-Interfaces ist. Rote Elemente bzw. Element mit der „−−“-Annotation sind ausschließlich Teil der *LHS* und werden im Rahmen der Regelanwendung im Modell(-graphen) gesucht und gelöscht. Grüne Elemente bzw. Elemente mit der „++“-Annotation beziehen sich ausschließlich auf die *RHS*, repräsentieren also neu zu erzeugende Elemente. Schwarze Elemente beziehen sich auf das Interface und damit sowohl auf die *LHS* als auch auf die *RHS*.

In Abbildung E.6 sind diejenigen Klassen des *SDM*-Metamodells abgebildet, die *GT*-Regeln in *SDM* beschreiben. Story-Patterns umfassen somit zwei Arten von Elementen, nämlich 1.) *Object-Variablen* (Klasse `ObjectVariable`) und 2.) *Link-Variablen* (Klasse `LinkVariable`). Erstere repräsentieren Knoten und letztere die Kanten der *GT*-Regeln. Beide Variablenarten werden nun genauer betrachtet.

Object-Variablen

Einerseits definieren Object-Variablen (*OVs*), welche Knoten das Graphmuster umfasst, falls sie sich auf die *LHS* beziehen. Andererseits legen sie fest, welche Knoten respektive Objekte im Modell neu anzulegen sind, falls sie sich ausschließlich auf die *RHS* der Regel beziehen. Innerhalb des Story-Patterns werden *OVs*, analog zur *UML*-Darstellung für Objekte in Objektdiagrammen, als Rechtecke mit der Beschriftung „<Objektname>:<Klassenname>“ visualisiert. Sie besitzen einen Typ, der sich im Klassennamen

widerspiegelt und sich in der abstrakten Syntax anhand der `type`-Referenz, vgl. Abbildung E.1, ergibt. Der Typ einer *OV* legt fest, zu welchen Objekten im Modellgraph sie „passt“. Damit wird die Menge aller Knoten im Modell unterteilt in kompatible und auszuschließende Kandidaten. Eine *OV* kann nur an solche Objekte gebunden werden, deren jeweiliger Typ entweder mit dem Typ der *OV* identisch ist oder es sich dabei um einen Untertypen im Sinne der Vererbung handelt.

Auch besitzen *OVs* einen Namen, der innerhalb eines Story-Patterns eindeutig sein muss. Für Namensliterale gelten die üblichen Java-Konventionen für (nichtstatische, lokale) Variablenbezeichner. Insbesondere Java-Schlüsselworte dürfen nicht verwendet werden, da dies in Fehlern im Generat resultieren würde. *OVs* die in verschiedenen Story-Patterns eines Diagramms enthalten sind und den gleichen Namen besitzen, stehen miteinander in Beziehung und sind miteinander „verschränkt“. D. h. namensgleiche *OVs* sind i. d. R. nicht unabhängig voneinander zu sehen. Die genaue Bedeutung einer *OV* bestimmt sich durch drei wesentliche Eigenschaften, die nun einzeln beschrieben werden.

Bindungszustand Eine *OV* kann einen von zwei möglichen *Bindungszuständen* annehmen, vgl. hierzu den Aufzählungstyp `BindingState` im *SDM*-Metamodell, Abbildung E.6. Möglich sind die Werte *gebunden* (`BOUND`) oder *ungebunden* (`UNBOUND`). Visuell unterscheiden sich ungebundene von gebundenen *OVs* durch eine dünnere gegenüber einer fetten Umrandung. Ungebundene bzw. freie Variablen stehen für normale, meist zu suchende Knoten einer *GT*-Regel. Falls sie sich nur auf die *LHS* oder das Interface beziehen (rote oder schwarze *OVs*), repräsentieren sie Knoten, die im Modellgraphen gesucht werden müssen. Beziehen sie sich dagegen exklusiv auf die *RHS* der Regel (grüne *OVs*), so stehen sie für Objekte, die im Modell durch die Regelanwendung anzulegen sind. Eine gebundene Variable steht dagegen für ein festes und konkretes Objekt aus dem Modellgraphen, das bereits anderweitig identifiziert oder vorgegeben wurde. Solch gebundene Variablen bilden die Start- bzw. Einstiegspunkte bei der Suche eines Treffers für des vollständige Muster. Somit muss pro Story-Pattern mindestens eine gebundene Variable existieren, falls im Modell Kontextelemente gesucht werden sollen. Auch muss für jede ungebundene *OV*, die Bestandteil der *LHS* (oder optional anzulegen) ist, ein *zulässiger Pfad*[29] zu mindestens einer gebundenen Variable existieren. Besagte Pfade beziehen sich dabei *nicht* auf den Kontrollflussgraphen sondern auf den Graphen der *GT*-Regel. Ein einzelner Pfad lässt sich also als alternierende Sequenz von *OVs* und *LVs* auffassen, der seinen Anfang bei einer gebundenen *OVs* nimmt und azyklisch ist. Grüne, als ungebunden modellierte Variablen oder optionale Variablen – *OVs* also, die sich also ausschließlich auf die *RHS* der Regel beziehen oder deren Entsprechung im Modell nicht zwingend vorausgesetzt werden – dürfen in zulässigen Pfaden nicht auftreten, da für sie zum Zeitpunkt der Mustersuche (noch) keine Elemente im Modell existieren (können). Grundsätzlich stellt diese Bedingung sicher, dass eine lokale Suche nach allen zu suchenden Elementen des Graphmusters, bei der ein initialer Treffer schrittweise durch Traversierung zu benachbarten Elementen erweitert wird, grundsätzlich möglich ist.

Bindungsoperator Ob sich eine *OV* ausschließlich auf die *LHS*, die *LHS und RHS* oder ausschließlich auf die *RHS* bezieht, wird durch den Wert des *Bindungsoperators*

[29] Die Suche muss entlang eines solchen Pfades auch möglich sein, was nicht immer gegeben ist.

festgelegt, vgl. hierzu den Aufzählungstyp `BindingOperator` im *SDM*-Metamodell, Abbildung E.6. Die Bedeutung der Literale ergibt sich wie folgt:

`DESTROY` – Die *OV* ist ausschließlich Teil der *LHS*. Das entsprechende Objekt im Modellgraphen wird entweder bei der Mustersuche gesucht oder wurde bereits gebunden. Das Modellelement wird im Zuge Regelanwendung gelöscht.

`CHECK_ONLY` – Die *OV* ist sowohl Teil der *LHS* als auch der *RHS*. Ein entsprechend konfigurierter Objektknoten gehört ebenfalls zum Muster und damit zum Kontext des potentiell vorhandenen schreibenden Anteils der Regel. Das durch die *OV* referenzierte Objekt wird entweder gesucht (ungebunden) oder ist bereits bekannt (gebunden). Durch diese Variable wird weder ein neues Objekt angelegt noch gelöscht. Wie wir später sehen werden, können aber Attribute des Objektes im Zuge der Regelauswertung modifiziert werden. Kann während der Mustersuche kein Objekt für eine solche Variable identifiziert werden, bricht die Mustersuche ohne vollständigen Match ab.

`CREATE` – Die *OV* ist ausschließlich Teil der *RHS*. Ein Objekt vom referenzierten Typ der *OV* wird bei der Regelanwendung erzeugt und ggf. mit anderen Knoten der Regel verbunden. Es existiert eine Besonderheit bei nur bedingt anzulegenden Knoten, auf die in Kürze noch eingegangen wird.

Bindungssemantik Die *SDM*-Sprache enthält Konstrukte, die ihre Verwendung angenehmer und entsprechende Diagramme kompakter machen sollen, indem die Art der Modellierung „deklarativer“ erfolgt. Grundsätzlich würde der Kontrollfluss (mit Sequenzen, Schleifen und Fallunterscheidungen) im Kombination mit sehr einfach gehaltenen Anweisungen bzw. Ersetzungsregeln bereits ausreichen, um jedes berechenbare Problem zu lösen. Allerdings ist leicht nachvollziehbar, dass auf komplexere Fallunterscheidungen mit Hilfe des Kontrollflusses möglichst zu Gunsten einzelner deklarativer *GT*-Regeln verzichtet werden sollte, da Letzteres gleichermaßen eleganter und verständlicher zu sein verspricht. So bietet die *SDM*-Sprache mit *optionalen* Variablen und mit *NAC*-Variablen entsprechende Sprachmittel an. Eine *OV* wird dazu mit Hilfe der Literale des Aufzählungstyps `BindingSemantic` des *SDM*-Metamodells, vgl. Abbildung E.6, entsprechend konfiguriert.

Eine *optionale OV* trägt das `BindingSemantics`-Literals `OPTIONAL`. Dadurch wird ausgedrückt, dass entsprechende Knoten des Musters, falls im Modell Entsprechungen existieren, bei der Regelausführung auch berücksichtigt werden. Sollten die entsprechenden Elemente im Modell jedoch fehlen, kann die Regel auch ohne diese Musteranteile ausgeführt werden. Jeder optionale Knoten ist dann im Prinzip eine Schreibabkürzung für zwei Varianten der Regel – einmal mit und einmal ohne die optionale *OV* (und möglicherweise deren Verbindungen zum Rest der Regel) – die als entsprechende Teilstrukturen in den Kontrollfluss einzubetten wären. In der Visualisierung sind optionale *OVs* durch einen gestrichelten Rahmen erkennbar.

Die zuvor angedeutete Besonderheit bei optionalen *OVs* besteht nun darin, dass optionale *OVs*, die gleichzeitig das `BindingOperator`-Literal `CREATE` tragen, nur dann erzeugt werden, wenn sie zuvor noch nicht existierten. Dazu ist es allerdings nötig, dass bereits vor der Erzeugung ein passendes Objekt vergeblich gesucht wurde. Somit haben entspre-

chend konfigurierte *OVs* auch einen Bezug zur *LHS* der Regel, obwohl sie sich durch die Wahl des Bindungsoperators eigentlich auf die *RHS* beziehen.

Soll die Existenz bestimmter Objekte im Graphen für eine erfolgreiche Regelanwendung ausgeschlossen werden, bietet die *SDM*-Sprache dafür das Sprachmittel der *NAC-OVs* an. Entsprechend modellierte Knoten werden im Rahmen der Mustersuche ebenfalls im Modell gesucht. Sollte sich ein Treffer m für das Muster *ohne* *NAC*-Knoten zu einem Treffer m'_i für eines der n „erweiterten Muster“ – jeweils ergänzt um eine einzelne, nun als obligatorisch angenommene Variante eines der n ursprünglichen *NAC*-Knoten – vervollständigen lassen (indem der Treffer m nur noch um das Objekt für den *NAC*-Knoten aus m'_i ergänzt wird), so stellt m selbst keinen Treffer des Ausgangsmusters mit *NAC*-Elementen dar, ansonsten schon. In anderen Worten, damit das vollständige Muster zu einem validen Treffer führt, muss das um die *NAC*-Elemente bereinigte Muster zu einem Treffer führen, der sich *nicht* zu einem Treffer für das bereinigte Muster *plus* eines der ursprünglichen *NAC*-Elemente (dann ohne als *NAC*-Elemente konfiguriert zu sein) ergänzen lässt.

In der abstrakten Syntax ist eine *NAC-OV* dadurch erkennbar, dass sie das `Binding Semantics`-Literal `NEGATIVE` trägt. In der konkreten Syntax sind entsprechende *OVs* „durchgestrichen“ dargestellt, vgl. z. B. die *OV* `tmpExpr` im Story-Knoten `add_as_first _param` in Abbildung A.13.

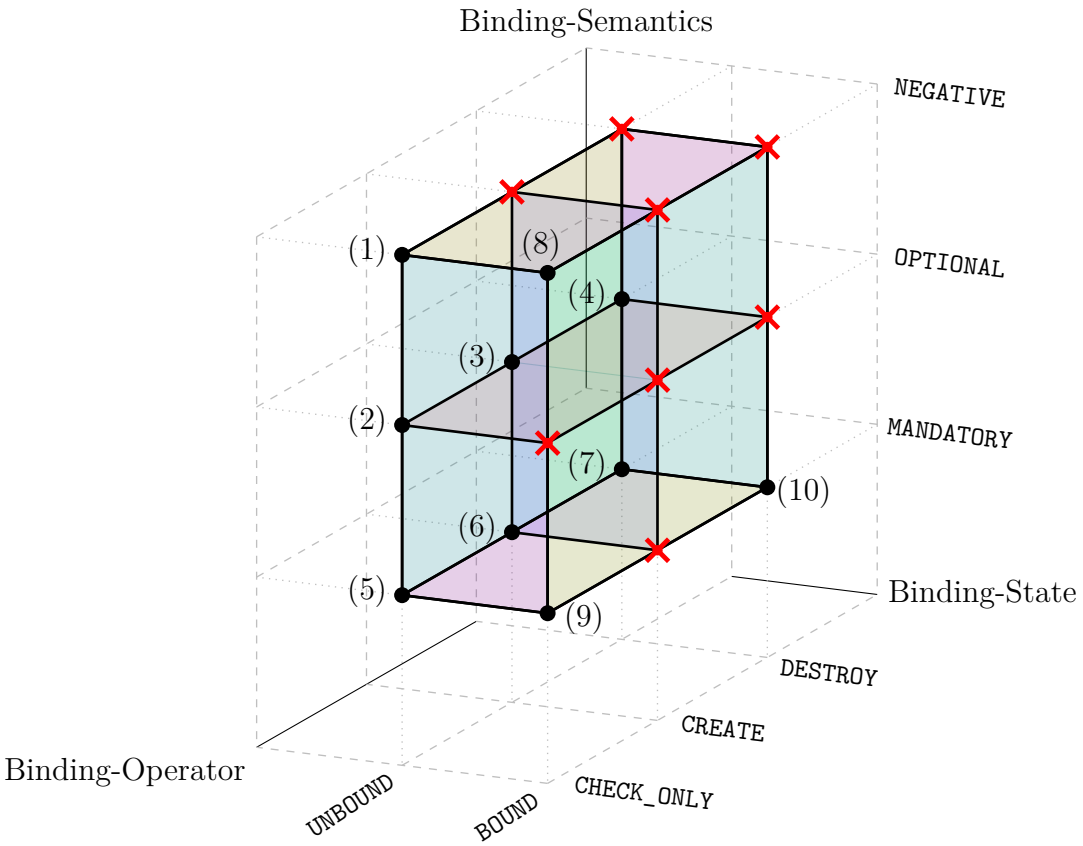


Abbildung 4.4: Eigenschaften von Object-Variablen und deren Ausprägungen

In Abbildung 4.4 sind die möglichen Kombinationen der drei *OV*-Eigenschaften Bin-

dungszustand, -operator und -semantik visualisiert. Nicht alle möglichen Kombinationen sind dabei sinnvoll und so markieren rote Kreuzchen invalide Konfigurationen. Ohne dass hier auf alle ausgeschlossenen Kombinationen dediziert eingegangen werden soll, werden einige Konstellationen exemplarisch betrachtet. Ist beispielsweise eine *OV* als gebunden modelliert, so ist der Wert `OPTIONAL` für die Bindungssemantik ausgeschlossen. Denn eine gebundene Variable kann hier nicht „optional gebunden“ sein – entweder sie ist es, oder sie ist es nicht. Auch können *NAC*-Knoten nicht mit dem Operator `DESTROY` kombiniert werden, da deren zu suchende Entsprechung im Modell ja nie Teil des eigentlichen Treffers sein kann. Auch die zwei Fälle für die Kombination aus `NEGATIVE` und `CREATE` werden ausgeschlossen, da diese semantisch keinen Sinn ergeben.

Die verschiedenen möglichen Kombinationen bei einer einzelnen *OV* sind noch mal in Tabelle 4.1 zusammengefasst, jeweils unter Angabe der jeweiligen konkreten Syntax. Die Nummerierung aus der Tabelle entspricht derjenigen aus Abbildung 4.4. In der Spalte „Konfiguration“ steht „st.“ für Binding-State, „op.“ für Binding-Operator und „sem.“ für Binding-Semantics.

Nr.	*Konfiguration*			*Semantik*	*Syntax*
	st.	*op.*	*sem.*		
(1)	u	c/o	neg	auszuschließendes NAC-Objekt	ov1 : SomeType
(2)	u	c/o	op	optionales Objekt	ov1 : SomeType
(3)	u	c	op	optional anzulegendes Objekt	++ ov1 : SomeType
(4)	u	d	op	optional zu löschendes Objekt	-- ov1 : SomeType
(5)	u	c/o	man	zu suchendes Objekt	ov1 : SomeType
(6)	u	c	man	anzulegendes Objekt	++ ov1 : SomeType
(7)	u	d	man	zu löschendes Objekt	-- ov1 : SomeType
(8)	b	c/o	neg	bereits gebundenes, identifiziertes Objekt (mit Bedingung) als NAC	ov1 : SomeType attr == aValue
(9)	b	c/o	man	bereits gebundenes, identifiziertes Objekt	ov1 : SomeType
(10)	b	d	man	bereits gebundenes, zu löschendes Objekt	-- ov1 : SomeType

Tabelle 4.1: Übersicht zu möglichen *OV*-Ausprägungen (Ergänzung zu Abbildung 4.4)

This-Variable Es gibt bei *SDMs* eine spezielle *OV*, die sich durch den reservierten Namen „this“ auszeichnet. Sie dient einem ähnlichen Zweck, wie das `this`-Schlüsselwort in Java und steht immer für das konkrete Objekt auf dem die (per *SDM* beschriebene) Operation zur Laufzeit gerade aufgerufen wird. Die spezielle *This-OV* besitzt als nominalen Typ folglich stets die Klasse, der die Operation im Metamodell ursprünglich zugeordnet

wurde. Sie ist immer als gebunden anzusehen (selbst bei der erstmaligen Verwendung im Kontrollfluss) und wird in der konkreten Syntax entsprechend als gebundene *OV* dargestellt. In der konkreten Syntax trägt diese spezielle *OV* immer den String „this“. Die Angabe des Typs könnte entfallen (Darstellung in MOFLON), wird hier allerdings mit aufgeführt (eMoflon-Darstellung).

Ansonsten lässt sich die *This-OV* genauso verwenden, wie andere *OVs* auch. Soll die *This-OV* allerdings innerhalb von Ausdrücken (im Folgenden noch beschriebenen) verwendet werden, z. B. als Parameter eines Operationsaufrufs, so muss sie aufgrund technischer Besonderheiten zuvor mindestens einmal in einem der Story-Patterns des Diagramms benutzt worden sein. Hierzu hat es sich etabliert, die *This-OV* beispielsweise im ersten Story-Pattern eines *SDM*-Diagramms (mit) aufzuführen.

Explizites Binden und Typecasting Durch *Bindungsanweisungen* können (bis dahin vorzugsweise ungebundene) *OVs* explizit an bestimmte Objekte gebunden und dadurch initialisiert werden. Hierdurch ist es möglich, dass eine neu angelegte *OV* anhand des Rückgabewertes einer Hilfsmethode (durch eine entsprechende Anweisung für einen Operationsaufruf) gebunden wird. Auch kann man hierdurch eine *OV* an ein spezifisches Objekt des Modells binden, welches bereits durch eine andere *OV* referenziert wird. So ist es möglich, ein referenziertes Objekt beispielsweise zwischenzuspeichern. Die erste Variante ermöglicht das Binden anhand des Ergebnisses einer komplexen Berechnung. Die zweite Variante bietet, neben dem Zwischenspeichern, auch die Möglichkeit zur Typanpassung im Sinne eines *Type-Casts*. Der Typ der explizit zu bindenden *OV* muss nämlich nicht identisch sein zu dem Typen der bestehenden *OV* (oder dem Rückgabetypen der Methode). Hierdurch wird auch mittelbar[30] eine Fallunterscheidungen anhand des aktualen Typs des referenzierten Objekts (bzw. des Operationsergebnisses) möglich. War eine Bindungsanweisung erfolgreich (und war das restliche Muster entweder leer oder wurde hierfür ein valider Treffer gefunden), so ist der aktuale Typ kompatibel zum nominalen Typ der neuen *OV*. Zum Testen auf Typkompatibilität ist allerdings ratsam, dass das entsprechende Story-Pattern nur die Bindungsanweisung beinhaltet. Nur so ist im Falle einer negativen Auswertung des Musters sichergestellt, dass die Ursache ausschließlich darin lag, dass der Type-Cast fehlgeschlagen ist (was den Sonderfall beinhaltet, dass sich der Wert der Zuweisung zu `null` ergibt).

Bindungsanweisungen können nur für als gebundene und obligatorisch modellierte *OVs* angegeben werden. In der konkreten Syntax wird eine Bindungsanweisung innerhalb des umrandeten Kästchens der *OV* durch eine spezielle Beschriftung der folgenden Form kenntlich gemacht:

```
<Variable_Name> ':' <Variable_Type> ':=' <Binding_Expression_String>
```

Entsprechende Beispiele sind den Abbildungen A.11 (`check_for_root_system`), A.12 (`check_if_aSystem_is_SubSystem`), A.13 (`try_to_cast_to_SumExpr`), A.15 (`check_if_assignment`) und A.18 (diverse) für den Type-Cast-Fall sowie den Abbildungen A.17 (`retrieve_topmost_assignment`), A.19 (`retrieve_topmost_assignment`), A.21 (`get_topmost_assignment`) und A.22 (`get_topmost_assignment`) für die *OV*-Bindung mittels Operationsaufruf zu entnehmen.

[30] Ein zum Java-Operator `instanceof` vergleichbares Konstrukt existiert in der *SDM*-Sprache nicht.

Attribute Klassen in *EMF*-Metamodellen umfassen i. d. R. Attribute, so dass Instanzen mit individuellen Wertebelegungen möglich sind. Eine praktisch nutzbare Transformationssprache sollte diese berücksichtigen. Insbesondere sind Fallunterscheidungen anhand von Attributwerten oder aber Zuweisungen/Änderungen von Attributen Grundvoraussetzung für die Erstellung sinnvoller Transformationen.

Bei der *SDM*-Sprache besteht die Möglichkeit, sowohl für gebundene als auch für ungebundene *OVs* Attributbedingungen zu formulieren, die dann im Rahmen der Mustersuche mit überprüft werden. Hierfür stehen einige Vergleichsoperatoren zur Verfügung: `"=="` und `"!="` und, für Attribute numerischer Art, zusätzlich `"<="`, `">="`, `">"`, `"<"` (die jeweilige Semantik ergibt sich analog zu den jeweiligen Java-Pendants). Auf den involvierten Objekten müssen alle vorhandenen Attributbedingungen einer Regel erfüllt sein, damit es sich um einen vollständigen Treffer handelt.

Attributwerte können als Teil der Regelanwendung auch modifiziert werden. Hierfür bietet *SDM* den Zuweisungsoperator `":="`. Nach der erfolgreichen Mustersuche werden im Zuge des Graphrewriting-Schritts allen zu modifizierenden Attributen die entsprechenden neuen Werte zugewiesen. Der Wert, mit dem der bestehende Wert eines Attributs verglichen wird sowie der neue Wert des Attributs (bei einer Zuweisung) werden jeweils durch die Angabe eines Ausdrucks, wie sie im nachfolgenden Abschnitt 4.3.3 beschrieben werden, festgelegt. So ist es beispielsweise auch möglich, dass sich der Wert einer Zuweisung aus dem Wert eines anderen Attributs ergibt. Komplexe Abhängigkeiten zwischen Attributwerten – beispielsweise durch Einbeziehung von Attributwerten unmittelbar vor und nach einem Rewriting-Schritt – die ein deklaratives *Constraint Satisfaction Problem (CSP)* aufspannen, vgl. hierzu beispielsweise [AVS12], werden hierbei zurzeit nicht unterstützt.[31]

Link-Variablen

Eine *Link-Variable (LV)* steht innerhalb eines Story-Patterns für eine *Kante* der *GT*-Regel und repräsentiert somit eine auf Existenz hin zu überprüfenden oder zu modifizierende Referenz im Modell (bzw. bezieht sich auf eine Java-Referenzen auf Ebene des Generats). *LVs* werden im *SDM*-Metamodell durch die Klasse `LinkVariable` berücksichtigt, s. Abbildung E.6. Die beiden wichtigsten Eigenschaften einer *LVs* sind die beiden *OVs*, die ihre *Quelle* und ihr *Ziel* darstellen. Quelle und Ziel sind dabei in der überwiegenden Mehrzahl der Fälle als nicht identisch anzunehmen, auch wenn dies nicht generell ausgeschlossen ist. Wie auch *OVs* besitzen *LVs* mit den Attributen `bindingOperator` sowie `bindingSemantics` Eigenschaften, die ihre Semantik im Rahmen der Regelauswertung stark beeinflussen. Die Implikationen sind ähnlich zu denen bei *OVs* und werden weiter unten genauer erklärt. Darüber hinaus verfügen *LVs* ebenfalls über einen Namen (eine Eigenschaft, die sie von der Oberklasse `NamedElement` erben). Anhand des Wertes dieses Namensattributs, welcher per Konvention zwingend dem Bezeichner eines Assoziationsendes im Metamodell entsprechen muss – auch muss eines der Assoziationsenden auch für mindestens einen Typ der beiden verbundenen *OVs* sichtbar sein – bestimmt sich der Typ einer *LV* implizit zu der passenden Assoziation.

[31] An einer entsprechenden Unterstützung für eMoflon wird aktuell geforscht.

Bindungsoperator Für den *Bindungsoperator* sind bei *LVs* ebenfalls, wie auch schon bei *OVs*, die drei Literale `CHECK_ONLY`, `CREATE` sowie `DESTROY` möglich. Die Bedeutung ergibt sich analog: Bei `CHECK_ONLY` muss der vorgegebene Link im Modell für einen Match vorhanden sein. Bei `CREATE` wird ein neuer Link im Zuge einer Regelanwendung zwischen den beiden referenzierten Objekten an den Enden angelegt, ohne dass dabei Annahmen über die Existenz anderweitiger Links getroffen werden. Und bei `DESTROY` wird der entsprechende Link im Modell gesucht und nach der Mustersuche im Rahmen der Regelanwendung gelöscht. Aufgrund der zugrunde liegenden *EMF*-Modellierung besteht allerdings eine Besonderheit bezüglich schreibender *LVs* im Falle von Containment-Beziehungen. So führt das „Setzen“ eines Links, der zu einer Containment-Beziehung im Metamodell gehört, dazu, dass eine eventuell zuvor bestehende Containment-Beziehung als impliziter Seiteneffekt aus dem Modell verschwindet. Ursächlich hierfür ist der Sachverhalt, dass in *EMF* nur maximal ein Container pro Objekt zu einem Zeitpunkt zulässig ist und das Rahmenwerk dies durchsetzt. Auch besteht bei einem Löschen eines solchen Links die Gefahr, dass einzelne Objekte bzw. Mengen von Objekten im Modell entstehen, die sich außerhalb der *EMF*-typischen Containment-Hierarchie befinden. Dies hat potentiell negative Auswirkungen für die weitere Verarbeitung des Modells. Insbesondere beim Serialisieren solcher Modelle können Teile nicht mehr erreicht werden. Auch ein ansonsten mögliches Navigieren entlang von (beliebigen) Kanten auch entgegen ihrer modellierten Navigationsrichtung, z. B. während der Mustersuche, kann hierdurch unmöglich werden, da die hierfür genutzte Hilfsfunktionalität auf eine konsistente Containment-Hierarchie angewiesen ist.

Die Wahl des Bindungsoperators einer *LV* kann nur unter Berücksichtigung der Bindungsoperatoren der beteiligten *OVs* erfolgen. Prinzipiell gilt dieser Zusammenhang auch in entgegengesetzter Richtung, allerdings werden typischerweise zuerst die *OVs* angelegt und konfiguriert, bevor die verbindenden *LVs* eingeführt werden. In Tabelle 4.2 sind die möglichen Fälle valider Kombinationen aufgeführt. Symmetrische Fälle wurden ausgespart (dunkelgraue Bereiche). Die unter (1) erwähnte Alternative ist in der verwendeten eMoflon-Version von Anfang 2014 im Zusammenspiel mit Containment-Kanten von Vorteil, da hierdurch effektiv verhindert werden kann, dass eine solche Kante „zu früh“ im Graphersetzungsschritt gelöscht wird (sondern erst implizit als Seiteneffekt). Deshalb wird diese Alternative (als notwendige aber suboptimale weil verwirrende Übergangslösung[32]) in der Beispieltransformation aus Anhang A.3 verwendet.

Die drei wichtigen Optionen, in denen die beteiligten *OVs* nicht modifiziert werden sollen, sind in der Tabelle mit (a)-(c) markiert. Sie repräsentieren die Fälle, dass (a) ein Link im Modell gesucht, und im Rahmen der Graphersetzung entfernt werden soll, (b) die Existenz eines Links im Modell nur als Vorbedingung der Regelanwendung überprüft werden soll, und (c) zwei bereits im Modell vorhandene Objekte durch einen neuen Link verbunden werden sollen.

Bindungssemantik Die Spezifikation einer *LV* umfasst auch die Angabe zur *Bindungssemantik*. Auch hier lauten die Optionen wieder `MANDATORY`, `OPTIONAL` und `NEGATIVE`. Da *LVs* keinen Bindungszustand besitzen, ihr (Ab-)Bild im Modell also nicht vorgegeben werden kann, ergibt sich der vollständig Zustandsraum einer *LV* gemäß Abbildung 4.5. Unmögliche Kombinationen sind durch Kreuze gekennzeichnet. Die konkrete Syntax al-

[32] Arbeiten an einer verbesserten Compilerversion laufen parallel zu dieser Arbeit.

Link-Variablen-Operator		Object-Variable 1		
		DESTROY	CHECK_ONLY	CREATE
Object-Variable 2	DESTROY	DESTROY[(1)]	DESTROY[(1)]	---
	CHECK_ONLY	...	(a) DESTROY, (b) CHECK_ONLY, (c) CREATE	CREATE[(2)]
	CREATE	...	...	CREATE[(2)]

[(1)] CHECK_ONLY ist in der aktuellen Implementierung ebenfalls möglich

[(2)] CHECK_ONLY ist nicht erlaubt, könnte aber implizit und automatisch in den CREATE-Operator umgewandelt werden

Tabelle 4.2: Bindungsoperatoren einer *LV* in Abhängigkeit der *OV*-Operatoren

ler legalen Kombinationen ist in Tabelle 4.3 gezeigt. Auch die Bedeutung der einzelnen Optionen wird kurz erläutert. Die Nummerierung der Tabelle entspricht der aus Abbildung 4.5.

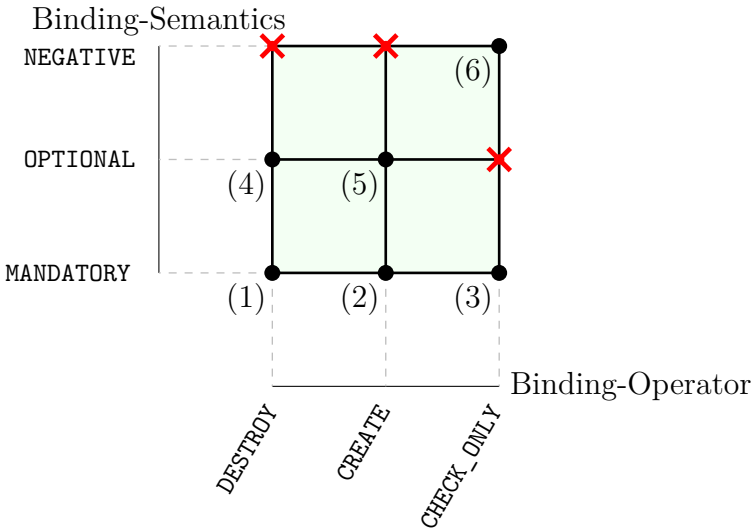


Abbildung 4.5: Eigenschaften von Link-Variablen und deren Ausprägungen

Navigierbarkeit Während der Mustersuche muss, wie bereits erklärt, für jede ungebundene *OV* eine Entsprechung im Modell gesucht werden. Dies setzt voraussetzt, dass für jede zu suchende *OV* mindestens ein gültiger (Such-)Pfad von einem gebundenen Knoten aus existiert. Da ein solcher Pfad nur anhand der tatsächlich in einem Muster vorhandenen Kanten gebildet werden kann, müssen auch entsprechende Kanten vorhanden sein. Dabei sind *LVs* hinsichtlich ihrer *Navigierbarkeit* bezogen auf den sog. *Suchplan*[33] in zwei Gruppen zu unterteilen. Es gibt navigierbare Links, die Teil eines entsprechenden Pfades sein dürfen, und nichtnavigierbare Links.

OVs, die als obligatorisch oder optional modelliert sind, sind beispielsweise nicht mittels optionaler Kanten zu binden, da ein vollständiger Match ja gerade auch ohne solche

[33] Zum Thema Suchplan s. z. B. [Zun96; VFV06]

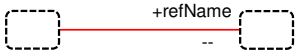
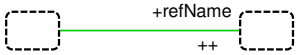
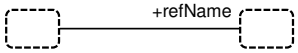
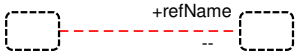
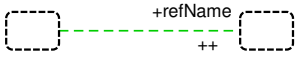
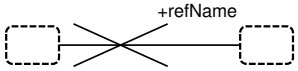
Nr.	Konfiguration		Semantik	Syntax
	op.	sem.		
(1)	d	man	zu löschende Kante	
(2)	c	man	anzulegende Kante	
(3)	c/o	man	zu suchende Kante	
(4)	d	op	optional zu löschende Kante	
(5)	c	op	optional anzulegende Kante	
(6)	c/o	neg	NAC-Kante	

Tabelle 4.3: Übersicht zu möglichen *LV*-Ausprägungen (Ergänzung zu Abbildung 4.5)

Kanten gültig sein soll. Auch ein Suchplan, der auf eine Traversierung von *NAC*-Kanten oder erst neu anzulegenden Kanten angewiesen wäre, ist als ungültig anzusehen. *NAC*-Kanten der Art (6) aus Abbildung 4.5 respektive Tabelle 4.3, die an *NAC*-*OVs* enden, sind hier ebenfalls ausgeschlossen (entweder ist die Kante als *NAC* spezifiziert, oder höchstens eines der beiden Enden). Entsprechende *NAC*-Kanten sind auch nur dann zulässig, wenn beide Enden durch einen validen Suchplan erreichbar sind (oder bereits gebunden sind). Als uneingeschränkt navigierbar zu betrachten sind dagegen *LVs* der Art (1) und (3).

4.3.3 Ausdrücke und die Schnittstelle zur Wirtssprache

Bisher wurden die Details der (textuellen) *Ausdrücke* (engl. Expressions) bewusst ausgespart. Nun werden diese nachgereicht. Dazu ist mit Abbildung E.8 ist eine entsprechende Sicht auf das *SDM*-Metamodell gegeben. Der Teilausschnitt des Metamodells beinhaltet alle relevanten Klassen aus den verschiedenen Paketen.

Anhand der acht angrenzenden Referenzen im Bereich `1`, die an der `Expression`-Klasse enden, sind alle Möglichkeiten zur Verwendung von Ausdrücken in *SDM*-Diagrammen abzulesen. Konkret können Ausdrücke in den folgenden Zusammenhängen auftreten:

- Als *Bindungsanweisung* für *OVs* (mittels `bindingExpression`-Referenz der Klasse `ObjectVariable`),
- Zur Definition von *Attributbedingungen* (per `constraintExpression`-Referenz der `Constraint`-Klasse),
- Zur Definition von *Attributzuweisungen* (mittels der `valueExpression`-Referenz von `AttributeAssignment`),
- Zur Formulierung des *Inhalts der Statement-Knoten* (mittels `statementExpression`-Referenz der Klasse `StatementNode`),
- Bei der Angabe des *Rückgabewertes bei Stopp-Knoten* (mittels `returnValue`-Referenz der Klasse `StopNode`),

- Bei der *Angabe einer aufzurufenden Methode* (mittels `target`-Referenz der `MethodCallExpression`),
- Zur Festlegung der *Parameterbelegung eines Methodenaufrufs* (mittels `valueExpression`-Referenz der Klasse `ParameterBinding`),
- Zur Angabe der beiden *Teilausdrücke einer Vergleichsanweisung* (mittels `rightExpression`- und `leftExpression`-Referenz der Klasse `BinaryExpression`).

Im Bereich der Abbildung E.8, der mit ② gekennzeichnet ist, sind darüber hinaus die speziellen Ausdrucksarten in Form der spezifischen `Expression`-Unterklassen angegeben. Diese werden nun individuell beschrieben.

MethodCallExpression

Method-Call-Anweisungen repräsentieren Operationsaufrufe und legen konkret fest, welche Operation auf welchem Objekt mit welchen Parametern aufzurufen ist. Dazu referenzieren sie eine (in einem der beteiligten Metamodelle modellierte) `EOperation` als Invokationsgegenstand (`callee`-Referenz). Die `target`-Referenz zeigt auf eine weitere `Expression`-Instanz, welche wiederum festlegt, auf welchem konkreten Objekt die angegebene `EOperation` aufzurufen ist. Das Objekt wird durch eine Instanz der Klasse `ObjectVariableExpression` oder der Klasse `ParameterExpression` festgelegt. Für den Fall, dass die aufzurufende Operation Parameter umfasst, referenziert die *Method-Call*-Anweisung zusätzlich noch eine Menge von `ParameterBinding`-Instanzen. Diese umfassen wiederum eigene Ausdrücke zur Definition der konkreten Parameterwertebelegungen.

Eine *Method-Call*-Anweisung ermöglicht es auch, (indirekt) beliebigen Java-Code aufzurufen und so innerhalb einer *SDM*-Transformation zu nutzen. Dazu muss ggf. noch eine Hilfsoperation im Metamodell eingeführt werden, deren Aufruf dann mit der Hilfe der Expression ausgelöst werden kann. Die Implementierung der zugehörigen Java-Methode (im Generat) erfolgt nicht durch die Codegenerierung für eine *SDM-GT* sondern durch manuelle Programmierung (so ist z. B. auch ein erneutes *Delegieren* an existierende, externe Funktionalität mit den Java-Sprachmitteln möglich). Der Codegenerierungsprozess von eMoflon ist dabei so flexibel, dass manuell eingebrachte Codefragmente im Generat (vorzugsweise durch sog. *Code-Injections*) auch bei wiederholtem vollständigen Neugenerieren erhalten bleiben.

ParameterExpression

Mit Hilfe von *Parameterausdrücken* (vgl. die Klasse `ParameterExpression`) kann auf die Werte einzelner Eingabeparameter der zur umgebenden *SDM*-Aktivität gehörenden Operation referenziert werden. Besitzt eine Operation beispielsweise einen Parameter, z. B. vom Typ `EBoolean`, könnte man dessen Wert so im Rahmen einer Attributszuweisung nutzen. Auch ein „Durchreichen“ von Parameterwerten an eine weitere, aufzurufende Operation ist eine häufig genutzte Verwendungsart solcher Anweisung.

AttributeValueExpression

Mittels *Attributwert*-Ausdruck (Klasse `ParameterExpression`) kann der Wert eines bestimmten Attributs einer *OV* referenziert werden. I. d. R. ist die referenzierte *OV* zu diesem Zeitpunkt bereits gebunden, was allerdings nicht zwingend der Fall sein muss.

So kann grundsätzlich auch auf Attribute einer im gleichen Muster zu bindenden *OV* zugegriffen werden (was allerdings potentiell zu Problemen mit der Suchplangenerierung führen kann). Dazu wird einerseits das entsprechende `EAttribute` mittels `attribute`-Referenz festgelegt, andererseits identifiziert die `object`-Referenz die entsprechende *OV*, deren Bild im Modell den konkreten Attributwert festlegt.

ObjectVariableExpression

Ein *Object-Variablen*-Ausdruck steht überall dort, wo der Wert der Belegung einer bestimmten *OV* benötigt wird. Ein entsprechender Ausdruck wird benutzt, um auf einem bestimmten Objekt eine Operation aufzurufen oder auch um Attributwerte des Objektes abzufragen. Funktional ist eine `ObjectVariableExpression` damit einer `Parameter Expression` sehr ähnlich. Statt allerdings den Wert eines bereits von Anfang an bekannten Parameters zu referenziert, wird hierdurch die erst noch zu etablierende Bindung einer *OV* abgefragt. Wurde ein Parameter in Form einer gebundenen *OV* in einer der *SDM*-Regeln verwendet, so lässt sich dessen Wert sowohl per `ObjectVariableExpression` als auch per `ParameterExpression` abfragen, wobei letzterer Variante der Vorzug gegeben werden sollte.

LiteralExpression

Die sogenannte *Literal-Expression* ist zur direkten Angabe von Literalen aus dem Repertoire der Wirtssprache (Java) bzw. von Literalen primitiver Typen (z. B. `EString`, numerische Typen) gedacht. Das entsprechende `value`-Attribut der Klasse wird dazu bei der Codegenerierung von eMoflon unmittelbar (und ohne tiefergehende Überprüfung) als Codefragment in den Quelltext übernommen. Dadurch ist es grundsätzlich auch möglich, auch wenn dringend davon abgeraten wird, quasi beliebigen Code in das Generat zu injizieren. Dies stellt allerdings eine Verletzung des Trennungsgebots dar und führt außerdem zu direkter Abhängigkeit von der Wirtssprache und der Codegenerierung.

ComparisonExpression

Vergleichsanweisungen sind die (zurzeit) einzigen konkreten *SDM*-Ausdrücke mit „zweiwertigem“ Charakter, also der Verknüpfung zweier Teilausdrücke dienen. Sie werden insbesondere bei der Formulierung von Attributbedingungen bei *OVs* benutzt. Mit ihrer Hilfe kann beispielsweise überprüft werden, ob ein `EString`-Attribut einen bestimmten Wert trägt, oder ausgedrückt werden, dass ein `EInt`-Attribut einen gewissen Wert nicht überschreiten soll.

Hierzu wird ein Ausdruck für den Attributwert mit einem zweiten Ausdruck, welcher den Vergleichswert festlegt, kombiniert. Die Art des Vergleichs wird durch den Wert des `operator`-Attributs der `ComparisonExpression`-Instanz definiert. Die entsprechenden Optionen wurden bereits bei der Beschreibung von *OV*-Attributen aufgeführt. Im *SDM*-Metamodell sind die Optionen als Literale der `ComparingOperator`-Enumeration erfasst, vgl. Abbildung E.5.

Eine weitere Verwendungsmöglichkeit für Vergleichsanweisungen liegt in der Bereitstellung des Wertes als Ziel einer Zuweisung. So kann ein Boolesches Attribut einer *OV* im Rahmen einer Zuweisung auf das Ergebnis eines Vergleiches einer (unabhängigen) Attribute-Value-Expression beispielsweise mit einer Literal-Expression festgelegt werden.

4.4 Eine vollständige Beispieloperation

Im Anhang A.3 ist die vollständige Implementierung einer nichttrivialen *SDM*-Beispieltransformation gegeben. In diesem Abschnitt wird die Transformation, die im weiteren Verlauf als Anschauungsobjekt und prototypisches Beispiel einer programmierten *GT* dient, beschrieben.

Gegenstand der Transformation ist in der Übersetzung von einfachen Blockdiagrammen, die durch das Metamodell aus Abbildung A.1 beschrieben werden – letzteres wurde bereits in Abschnitt 2.2.3 als Beispiel aufgeführt – in Instanzen des Metamodells nach Abbildung A.2, das wiederum den Aufbau einfacher Java-Klassen beschreibt.[34] Die eingegebenen Blockdiagramme modellieren einzelne arithmetische Berechnungsvorschriften (Funktionen), die aus Additions-, Multiplikations- und Verstärkungsoperationen aufgebaut sind. Auf der Ausgabeseite enthalten die Java-Modelle Beschreibungen von Mengen an Methoden und Feldern (jeweils als Teile einer Java-Klasse). In ihrer Gesamtheit kodieren die Java-Elemente die Berechnungsvorschrift als ausführbares Programm.

Die Modelltransformation besteht dabei aus zwei wesentlichen Teilfunktionalitäten. Diese sind separat umgesetzt mit Hilfe der beiden Klassen `Bd2JaConverter` und `BdPre processor`, welche beide Teile eines Hilfsmetamodells zur Transformationsbeschreibung darstellen, vgl. Abbildung A.3. Konkret handelt es sich bei den Teilfunktionalitäten einerseits um dem eigentlichen Übersetzungsschritt, der ein Blockdiagramm auf das Java-Modell abbildet, andererseits um einen vorgeschalteten Optimierungsschritt, in dem das Eingabeblockdiagramm normalisiert wird. Die Normalisierung stellt sicher, dass bestimmte Blockkonstellationen durch einfachere Strukturen ersetzt werden. Beide Teilaspekte werden im Folgenden kurz skizziert.

4.4.1 Normalisierung von Blockdiagrammen

Durch den Normalisierungsschritt sollen zwei Eigenschaften in der Ausgabe sichergestellt werden. Einerseits sollen im Anschluss nur noch `Add`- oder `Mult`-Blöcke/-Instanzen mit noch genau zwei `Inport`-Instanzen (und jeweils einem `Outport`) vorhanden sein. Dazu werden einzelne Blöcke mit mehr als zwei Eingabe-Ports (im Eingabemodell) durch schrittweises *Kaskadieren* in eine funktional äquivalente Darstellung übersetzt. Neben der Korrektur der Inport-Anzahl sollen andererseits auch alle ggf. vorhandenen `Gain`-Blöcke durch eine Darstellung auf Basis eines `Mult`- und eines `Constant`-Blocks ersetzt werden. Der ursprüngliche Verstärkungsfaktor wird in Form des `Constant`-Blocks berücksichtigt. Ein Beispiel für ein vollständiges Eingabemodell sowie die daraus resultierende Ausgabe der Normalisierungstransformation sind in Abbildung 4.6 gegeben. Das Beispiel umfasst alle drei Arten von Ersetzungen (Kaskadierung jeweils eines `Add`- und eine `Mult`-Blocks sowie Ersetzung von `Gain`-Instanzen).

Bei der Transformation handelt es sich um eine endogene und horizontale Inplace-Transformation. Sie wird monolithisch als Batch-Transformation ausgeführt und implementiert eine Art von Refactoring. Somit weist dieser Teil der Gesamttransformation aus technischer Sicht vielfach andere Eigenschaften auf als der nachgelagerte Übersetzungsschritt. Warum erscheint eine solche Normalisierungstransformation hier als so sinn-

[34] Es handelt sich dabei weder um eine vollständige bzw. exakte Beschreibung von Java noch um ein Metamodell für Java-*ASTs*.

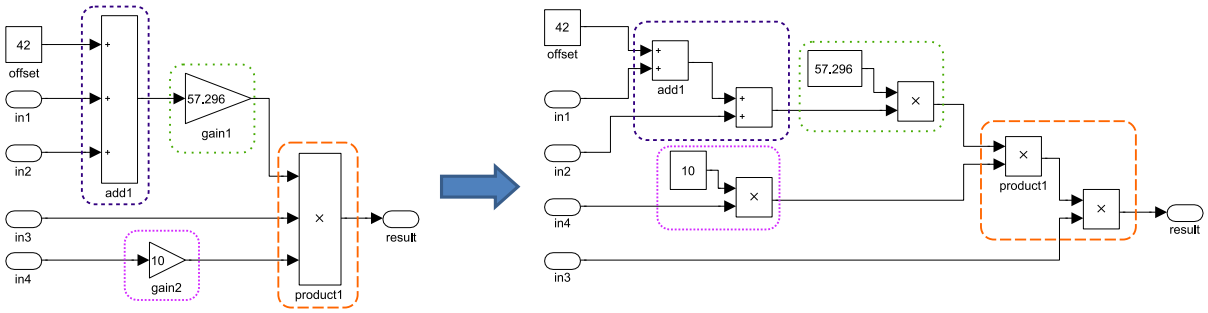


Abbildung 4.6: Die drei Operationen zur Normalisierung am konkreten Beispiel (sich jeweils entsprechende Teilstrukturen in Ein- und Ausgabe sind einheitlich umrandet)

voll, dass deren Implementierung als Teil dieser Arbeit entwickelt wurde?[35] Dazu sollte man sich verdeutlichen, dass durch diese Normalisierung der anschließende Übersetzungsschritt vereinfacht wird, da einige der sonst auftretenden Sonderfälle auszuschließen sind.

Ohne auf jedes kleine Detail der entsprechenden *GT*-Regeln eingehen zu wollen, soll nun die Struktur der Implementierung aus Anhang A.3.2 vorgestellt werden. Den Einstiegspunkt zur Normalisierung bildet die `process`-Operation der Klasse `BdPreprocessor`, s. Abbildung A.6. In ihrem ersten For-Each-Knoten werden alle `RootSystem`-Instanzen bestimmt und jede dieser im Rahmen der Each-Time-Komponente an die Hilfsoperation `collectRelevantBlocks`, s. Abbildung A.4, übergeben. In dieser werden dann alle `Add`- und `Mult`-Instanzen mit mehr als zwei Inports sowie alle `Gain`-Instanzen innerhalb des übergebenen Systems identifiziert und in die entsprechenden Listen der `BlockCollector`-Instanz eingetragen. Für eventuell vorhandene Subsystemblöcke wird dazu gehörende `SubSystem`-Instanz identifiziert und die aktuelle Operation mit dieser Instanz als Parameter erneut rekursiv aufgerufen. Nachdem alle zu normalisierende Teilstrukturen des Modells identifiziert wurden, werden im Anschluss innerhalb der fortgesetzten `process`-Operation zuerst alle identifizierten `Add`-Instanzen durch Aufrufe der `splitAdd`-Operation, vgl. Abbildung A.7, aufgespalten und danach die `Mult`-Instanzen durch jeweiligen Aufruf der Operation `splitMult`, vgl. Abbildung A.8. Zuletzt werden noch die `Gain`-Instanzen durch Aufrufe der `normalizeGain`-Operation aus dem Modell entfernt.

Die Implementierungen der `splitAdd`- und der `splitMult`-Operation basiert jeweils auf einem schrittweisen Aufspalten und Neuverbinden. Beide Implementierungen unterscheiden sich nur marginal voneinander (Unterschiede ergeben sich bzgl. der Verwendung des angepassten Blocktyps sowie der jeweils passenden rekursiven Aufrufe). Grundsätzlich sind diese beiden Implementierungen aufgrund ihrer relativ umfangreichen ersten *GT*-Regel interessant für den zu entwickelnden Testansatz, da sich aufgrund der höheren Komplexität auch mehr Raum für potentielle Fehler ergibt. Eine Besonderheit der Implementierung stellt die schwarze (Containment-)Kante zwischen den *OVs* `inport2` und `toBeSplit` in Kombination mit der grünen Kante gleichen Typs zwischen den *OVs* `inport2` und `newAdd` bzw. `newMult` dar. Eigentlich sollten die schwarzen Kanten hier rot sein, da sie durch die Anwendung der Regel entfernt werden sollen. Allerdings sorgen gewisse Eigenheiten der Codegenerierung dafür, dass es bei der eigentlich korrekten Mo-

[35] Ähnliche Beispieltransformationen wurden auch im MATE-Kontext verwendet, vgl. [Stu+07].

dellierungsvariante zu fehlerhaftem Verhalten kommt. Die hier gewählte Modellierung resultiert in korrektem Verhalten, da durch das Neuanlegen der jeweiligen Containment-Beziehung die alten Beziehungen als Seiteneffekt auch gelöscht werden.

Fehlt noch die `normalizeGain`-Operation aus Abbildung A.5. Sie verbindet den alten In- und den Outport des zu löschenden `Gain`-Blocks neu, so dass alle an diesen endenden `Line`-Instanzen erhalten bleiben. Ebenfalls aufgrund technischer Besonderheiten, muss hier aber das eigentliche Löschen der `gain`-OV verzögert in einem zweiten Schritt erfolgen.

4.4.2 Von Blockdiagrammen zu Java-Beschreibungen

Der zweite Teil der Blockdiagramm-zu-Java-Transformation (Bd2Ja) ist mit 14 Operationen, verglichen mit den fünf zuvor, komplexer. Auch im Hinblick auf wesentliche Eigenschaften bestehen Unterschiede zum Normalisierungsschritt; die Bd2Ja-Transformation ist eine exogene, vertikale Outplace-Übersetzung. Gemeinsam ist beiden Transformationen, dass es sich auch hier um eine monolithisch auszuführende, unidirektionale Batch-Transformationen handelt, und dass bei beiden das Eingabemetamodell identisch ist.

Ziel des Bd2Ja-Teils ist es, aus einer Blockdiagramm-Darstellung einer Berechnungsvorschrift eine Beschreibung einer Java-basierten Implementierung abzuleiten. Dabei werden Systeme auf Blockdiagrammebene zu Java-Methoden, In-Ports zu Methodenparametern, Out-Ports zu Rückgabewerten von Methoden und Blöcke für mathematische Berechnungen zu entsprechenden Teilausdrücken in Java. Im Hinblick auf realistische, praktische Transformationsaufgaben, könnte man sich eine solche Transformation als einen wesentlichen Teil eines domänenspezifischen Codegenerierungsansatzes vorstellen.

Einstiegspunkt der Transformation ist die Operation `convert(BDFile):JPackage` der Klasse `Bd2JaConverter`, s. Abbildungen A.9 und A.10. Diese ruft als erstes `init`, s. Abbildung A.16, auf – eine Übersicht der Zusammenhänge zwischen den einzelnen Operationen lässt sich auch dem Call-Graphen aus Abbildung A.23 entnehmen. Innerhalb von `init` werden zuerst (in den ersten vier Knoten) eventuell vorhandene Überbleibsel einer zuvor ausgeführten Abbildung entfernt. Anschließend wird eine `BdPreprocessor`-Instanz angelegt, falls diese noch nicht existiert (und dabei die OV `preProc` gebunden). Die unteren drei Knoten dienen der bedingten (Re-)Initialisierung der `BlockCollector`-Instanz, so dass keine ggf. vorhandenen, veralteten Verweise auf Blöcke die Ausführung stören können.

Nachdem `init` durchlaufen wurde, wird in `convert` als nächstes die übergebene `BDFile`-Instanz auf eine neu erstellte Instanz von `JPackage` abgebildet und der Name entsprechend gesetzt. Außerdem wird die Instanz von `BdPreprocessor` gesucht, die nach der Initialisierung vorhanden sein sollte. Auf dieser wird dann die `process`-Operation aufgerufen, was die zuvor beschriebene Normalisierungstransformation ablaufen lässt. Im Anschluss daran wird (als Sanity-Check) überprüft, ob im obersten System wirklich nur eine `Out`-Instanz vorhanden ist. Im Erfolgsfall werden alle `RootSystem`-Instanzen (die von der `BDFile`-Instanz referenziert werden) individuell durch eine Abbildung auf eine neue `JClass`- bzw. `JMethod`-Instanz übersetzt, s. den For-Each-Knoten `create_jclasses` sowie die Each-Time-Komponente in Abbildung A.10.

Im Rahmen der Übersetzung eines Root-Systems werden zuerst alle `Constant`-Instanzen in Java-Felder übersetzt. `In`-Blöcke repräsentieren Eingangsdaten der Berechnung, weshalb diese in Parameter der Java-Methode zu übersetzen sind. Wichtig ist dabei,

dass die Parameter eine unveränderliche (aber beliebige) Reihenfolge aufweisen. Dies wird durch den Aufruf der Hilfsoperation `establishJParamOrdering` sichergestellt, deren *SDM*-basierte Implementierung[36] in Abbildung A.14 gegeben ist.

Nachdem die Entsprechung für eine `RootSystem`-Instanz angelegt wurde, müssen auch eventuell vorhandene (und ggf. tiefer verschachtelte) `SubSystem`-Instanzen übersetzt werden. Diese Aufgabe wird von der `createSystemToMethodMappings`-Operation übernommen, die sich ggf. selbst rekursiv aufruft, s. den Statement-Knoten `recursive_call` in Abbildung A.12.

Sind alle `BDSystem`-Instanzen in ihre `JMethod`-Gegenstücke übersetzt, besteht der letzte Schritt der Each-Time-Komponente aus Abbildung A.10 darin, dass der eigentliche Inhalt der Systeme in Form der darin enthaltenen Blöcke transformiert wird. Dieser Prozess startet durch die Delegation an die `convertSystem`-Operation, deren Implementierung in Abbildung A.11 abgebildet ist.

Der eigentliche Übersetzungsvorgang läuft dann so ab, dass, ausgehend von dem einzelnen Ausgabeblock in dem jeweiligen (Sub-)System, das Teildiagramms rückwärts (entgegen der Richtung des Datenflusses) durchlaufen wird – durch entsprechendes Navigieren sowie wiederholtes Aufrufen der `visitBlock`-Operation für die angetroffenen Blöcke – bis letztendlich die Eingabeblöcke erreicht werden.[37] Innerhalb der `visitBlock`-Operation, vgl. Abbildung A.18, wird explizit (und rein mit *SDM*-Sprachmitteln) der Kontrollfluss anhand des aktualen Blocktyps so verzweigt, dass an eine passende Operation weiter delegiert wird.

Die Behandlung von `In`- und `Constant`-Blöcken unterscheidet sich nur marginal (anhand der durch die `...Expr`-Instanzen referenzierten Java-Modellelemente), wie durch einen Vergleich der *SDM*-Realisierungen aus den Abbildungen A.20 und A.19 nachvollzogen werden kann. Auch die Transformationen von `Add`-, Abbildung A.17, sowie `Mult`-Blöcken, Abbildung A.21, unterscheidet sich nicht fundamental. Für beide Operatoren wird eine entsprechende Instanz einer `BinOpExpr`-Unterklasse angelegt, die wiederum zwei Unterausdrücke für die beiden Operanden beinhaltet. Letztere ergeben sich durch erneute Aufrufe der `visitBlock`-Operation, jeweils mit einem der (mit den Eingängen) verbundenen Blöcke als Parameter.

Fehlt nur noch die Betrachtung der `visitSubSystemBlock`-Operation, abgebildet in Abbildung A.22. Die zweite Hälfte der Operation, unmittelbar im Anschluss an die `embed_call_expression`, ist etwas komplexer im Vergleich zur Übersetzung der `Add`- und `Mult`-Blöcke. Dies liegt daran, dass *SubSystemBlock*-Instanzen mehr als zwei Eingangsports besitzen können (denn dies wurde durch die Normalisierung ja nicht ausgeschlossen). Somit ist einerseits ein For-Each-Knoten (`iterate_through...`) nötig, andererseits muss im Anschluss auch der Inhalt des zugehörigen Subsystems berücksichtigt werden, weshalb an dieser Stelle auch ein erneuter Aufruf der `convertSystem`-Operation vorhanden ist.

[36] Eine direkte Implementierung mit Java wäre ebenfalls möglich.

[37] Die hier vorausgesetzten Annahmen über den Aufbau valider Eingabemodelle macht eine gesonderte Behandlung von Datenflussschleifen im Diagramm überflüssig. Auch um das Beispiel nicht noch weiter zu verkomplizieren, wurde deshalb auf die Behandlung solcher Schleifen verzichtet.

Testing shows the presence, not the absence of bugs.

(E. W. Dijkstra, aus [BR70])

5 Testen von Software

Wert und Nutzen einer Software bestimmen sich, neben ihrem Funktionsumfang, auch anhand ihrer Qualität. Letztere sollte der zugedachten Aufgabe angepasst sein, wie beispielsweise in der Einleitung von [Lig02] dargestellt wird. Die Qualität eines Softwaresystems ist nach [IEE90, S. 60] ein Maß dafür, in welchem Maß das System oder die betrachtete Komponente (i) die spezifizierten Anforderungen und (ii) die Erwartungen und Wünsche der Anwender erfüllt bzw. erfüllen kann. Weiterhin lässt sich der Qualitätsbegriff anhand verschiedener Qualitätsmerkmale konkretisieren. In [Lig02, Abb. 1.3] werden *interne*, *externe* und erst *zum Benutzungszeitpunkt* zum Tragen kommende Qualitätsmerkmale unterschieden, wobei auf den ISO/IEC-9125-Standard[1] als Primärquelle verwiesen wird. Bezogen auf den Inhalt dieser Arbeit sind vor allem die in [Lig02] aufgeführten Qualitätsmerkmale *Functionality* (funktionale Korrektheit, Genauigkeit der Ergebnisse) und, mit weniger Gewicht, *Safety* (Betriebssicherheit) von Interesse. Allerdings können auch relevante Anforderungen bezüglich *Reliability* (Robustheit, Fehlertoleranz), *Efficiency* (Effizienz im Sinne eines sparsamen Umgang mit Ressourcen) und – mit Abstrichen – Manipulationssicherheit (überlappt teilweise mit Aspekten funktionaler Korrektheit und auch Robustheit) bestehen.

Fehlerhafte Software kann ein erhebliches Sicherheitsrisiko[2] für Gesundheit, Leben und Umwelt darstellen. Offensichtlich ist dies für eingebettete Software, die in potentiell „gefährlichen“ Maschinen wie Autos, Flugzeugen etc. und Geräten wie Röntgenapparaten, Abfüllanlagen, Herzschrittmachern, Rüstungsgütern etc. zum Einsatz kommt. Aber auch Anwendungssoftware und insbesondere Entwicklungswerkzeuge (IDEs, Compiler, Codegeneratoren, Converter), die mittelbar einen großen Einfluss auf solch eingebettete Software haben können, sind ggf. selbst mit Risiken behaftet.

Aufgrund der zentralen Bedeutung von Software in vielen technischen Systemen und

[1] Der besagte Standard wurde durch den neueren Standard ISO/IEC 25000 abgelöst.

[2] Das Risiko steigt einerseits mit dem zu erwartenden Schaden, andererseits mit der Eintrittswahrscheinlichkeit des Ereignisses, vgl. z. B. [Eck03, S. 14].

Domänen unterliegt sie teils strengen Anforderungen bezüglich ihrer Zuverlässigkeit, Betriebssicherheit sowie Korrektheit hinsichtlich der Spezifikation. Andererseits setzen ökonomische und praktische Überlegungen natürliche Grenzen für die Überprüfbarkeit und Sicherstellung dieser Anforderungen, wobei sich Qualitätssicherungsmaßnahmen durchaus amortisieren können, falls durch sie kostspielige Fehlerkorrekturen oder Schäden verhindert werden.

Neben *organisatorischen* Maßnahmen wie Schulungen, Prozesse, unabhängige QS-Abteilungen etc. und *präventiv-konstruktiven* Maßnahmen wie „defensives“ Programmieren, statische Analysen oder die Nutzung geeigneter Programmiersprachen und Bibliotheken, Entwurfsmuster und Best-Practices etc., vgl. auch [Bei90; Bal98; Lig02], sind *überprüfende Verfahren* von zentraler Bedeutung für die Qualitätssicherung während und nach einzelnen Entwicklungsschritten oder -phasen. In Abbildung 5.1 sind konkrete Prüftechniken anhand einer detailreichen Klassifikation nach [Lig02, Abb. 1-13] zusammengefasst. Die Ansätze mit der größten Nähe zu den Beiträgen aus dieser Arbeit sind in der Abbildung mit Grau hinterlegt. Grundsätzlich werden neben den *verifizierenden*, im Sinne der *formalen Verifikation*, und den *analysierenden* Verfahren vor allem die *dynamisch-testenden* Verfahren in der Praxis wiederholt eingesetzt. Dabei wird das Testen häufig als *Verifikationstechnik*, das aber nicht zu verwechseln ist mit *formaler* Verifikation, gesehen. Es soll schließlich geprüft werden, ob die Implementierung zu ihrer Spezifikation „passt“, also ob das System *richtig* realisiert wurde. Die zentrale Fragestellung einer Validierung, nämlich ob das „richtige System“ entwickelt wird, hat beim Testen mit Blick auf die Korrektheit allenfalls eine geringe Bedeutung. Bei sog. Akzeptanztests ist dies ggf. anders einzuschätzen.

In der nachfolgenden Definition 5.1 wird der Begriff des Testens formal eingeführt.

Definition 5.1 (Testen, angelehnt an [IEE90]):

Den Prozess der Analyse von Software hinsichtlich
(a) Diskrepanzen zwischen gewünschten und realisierten funktionalen Eigenschaften,
(b) potentiell vorhandener Fehlern (Bugs),
(c) ihrer als wesentlich erachteten nicht-funktionalen Eigenschaften,
jeweils auf Basis einer ***dynamischen Ausführung*** *unter definierten (und kontrollierbaren) Bedingungen und der* ***Beobachtung ihres Verhaltens****, bezeichnen wir als* Testen*. Die Erstellung, Verwaltung und Dokumentation der dafür nötigen Artefakte sowie die Auswertung und Aufbereitung von Testergebnissen können ebenfalls als Teil des Testens verstanden werden.*

Das zu testende Softwaresystem, i. d. R. als *System Under Test (SUT)*[3] bezeichnet, wird also mit Hilfe *repräsentativer* (Test-)Eingaben bzw. Daten angeregt, und im Zuge dessen wird das dynamische Ausführungsverhalten beobachtet, untersucht und bewertet. Hierfür ist es wesentlich, dass das zu testende System bzw. sein Zustand und das Verhalten direkt oder indirekt beobachtbar ist, vgl. hierzu den Begriff der *Software Observability* von Ammann und Offutt [AO08, S. 14, Def. 1.11]. Je nach Eigenschaften des *SUT* sowie des Testansatzes ist entweder das gesamte Ausführungsverhalten inklusive des internen Systemzustands unmittelbar beobachtbar. In diesem Fall spricht man dann von sog. *Glass-Box*-Ansätzen. Falls die interne Struktur des Systems für den Entwurf der

[3] Alternativen nach [Bal98, S. 393]: *Prüfling*, *Testling* oder *Testobjekt*.

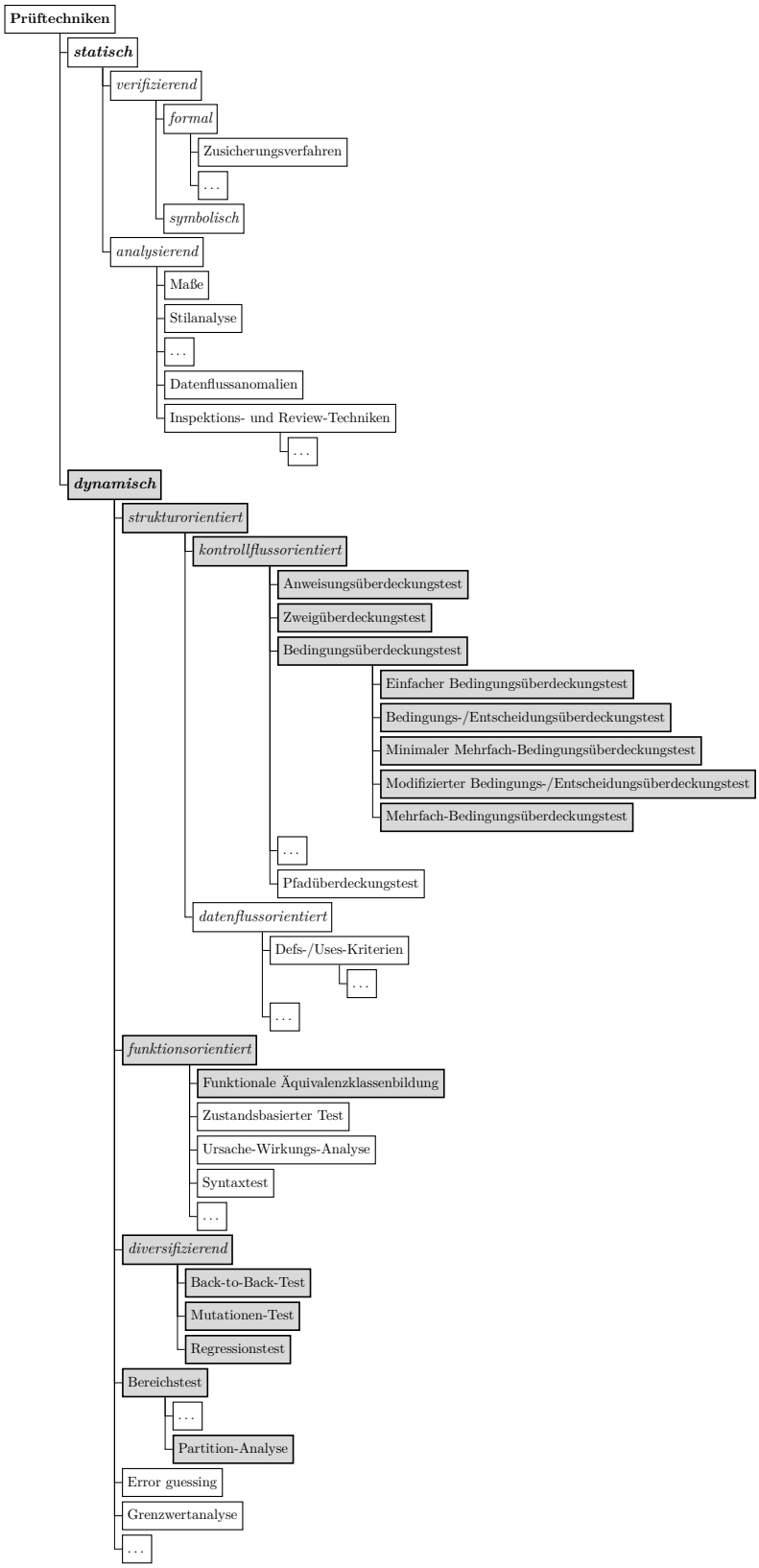


Abbildung 5.1: Klassifikation von Prüftechniken für Software (inkl. konkreter Nennungen) nach [Lig02, Abb. 1-13]

Testfälle herangezogen wird, handelt es sich um *White-Box*-Testen bzw. *strukturbasiertes* Testen, vgl. [Bei90, S. 2.2]. Oder aber das System kann anhand des Verhaltens an seinen Ausgängen z. B. in Form von Rückgabewerten einer Funktion, Methode oder Operation teilweise beobachtet werden. Man spricht in diesem Fall vom sog. *Black-Box*-Testen. Falls nur das Ein- und Ausgabeverhalten für den Testentwurf benutzt wird, spricht man vom *funktionsorientierten* Testen [Bei90]).

Das übrige Kapitel ist in fünf Unterabschnitte aufgeteilt. In Abschnitt 5.1 werden die verschiedenen mehr oder weniger naheliegenden Gründe für das Testen beleuchtet. Anschließend werden in Abschnitt 5.2 die wichtigsten Begriffe und Konzepte zum Testen eingeführt. Die verbleibenden drei Abschnitte sind konkreten Techniken gewidmet: in Abschnitt 5.3 werden einige klassische Überdeckungskonzepte vorgestellt, Abschnitt 5.4 präsentiert Mutationstestansätze und modellbasiertes Testen ist Thema von Abschnitt 5.5.

5.1 Ziele des SW-Testens

Tests können aus ganz unterschiedlicher Motivation erstellt und eingesetzt werden. Für eine abschließende Bewertung von Testverfahren ist es notwendig, dass das Ziel, welches durch das Testen werden soll, berücksichtigt wird. Im Folgenden werden einige typische Ziele für das Testen, im Hinblick auf nachzuweisende *SUT*-Eigenschaften, vorgestellt.

5.1.1 Steigerung des Vertrauens

Das vorrangige, abstrakte Ziel aller Testverfahren ist es, durch Beobachtung des Systems während seiner Ausführung unter repräsentativen, bekannt oder vermutet fehleranfälligen, kontrollierten Bedingungen das *Vertrauen* in das *SUT* zu steigern. Das ist wenig verwunderlich, da sich dies mit dem allgemeinen Ziel der Qualitätssicherung deckt, s. [Bal98, S. 278]. Der Vertrauensbegriff bezieht sich darauf, dass sich das *SUT* so verhält wie gewünscht, d. h. dass es sich konform zur Vorstellung bzw. zur Spezifikation verhält und dabei ggf. auch gewisse ihm zugedachte Eigenschaften aufweist. Da das Testen i. d. R. nur exemplarisch erfolgen kann – ein Testen mit *allen möglichen* Eingaben ist meist schon aufgrund ökonomischer Überlegungen ausgeschlossen – ist das Vertrauen stets mit einem gewissen (Rest-)Risiko behaftet.

5.1.2 Korrektheitstests

Das zweifelsohne wichtigste Ziel beim Testen ist die Überprüfung der *Korrektheit* eines *SUT*. Dabei verhält sich ein System genau dann *korrekt*, wenn es sich für seine *zulässige Eingaben* so verhält, wie dies durch eine zugrunde liegende *Spezifikation* vorgegeben wird, vgl. z. B. die Ausführungen zum Thema Korrektheit in [Lig02, S. 7 f.]. Korrektheit kann also immer nur im Sinne einer irgendwie gearteten, dokumentierten bzw. fixierten Vorstellung von einem *gewünschten Verhalten* bewertet werden. So wäre beispielsweise ein Programm, das mittels seiner Implementierung die Summe zweier Integer-Parameter berechnet und zurückliefert, eine korrekte Implementierung für eine Spezifikation eines Programms zur Berechnung der Summen zweier Elemente aus $\mathbb{N}$, aber inkorrekt für die Spezifikation eines Programms zur Berechnung des Produktes solcher Zahlen.

In welcher Form eine Spezifikation vorliegt, ob explizit – in Form eines in natürlicher oder formaler Sprache verfassten Dokuments – oder implizit als (geteilte) Vorstellung, ist weniger entscheidend. Wichtig ist, dass die durch die Spezifikation geforderten Eigenschaften des Systems bzgl. seiner Korrektheit widerspruchsfrei und objektiv überprüfbar sind. Es lassen sich grundsätzlich *syntaktische/statische* Korrektheit und *semantische* Korrektheit unterscheiden, vgl. hierzu z. B. [Ger+02].

Zur Überprüfung der Korrektheit wird das *SUT* mit *zulässigen* Eingaben stimuliert, die Systemreaktion beobachtet und anschließend unter Berücksichtigung der Vorgaben durch die Spezifikation bewertet.

5.1.3 Robustheitstests

Die *Robustheit* eines Systems beschreibt sein Fähigkeit, mit unerwarteten, falschen oder sehr vielen Eingaben sinnvoll umzugehen, ohne dadurch in einen undefinierten oder unzulässigen Zustand überzugehen, vgl. z. B. [Lig02, S. 10 f.]. Ein mögliches Ziel des Testens kann also sein, das System mit nichtspezifizierten, falschen und/oder sehr vielen Eingaben anzuregen, und so herauszufinden, ob das *SUT* robust genug ist. Da die Menge der möglichen Eingaben sehr groß oder sogar unbegrenzt sein kann, das System, falls zustandsbehaftet, darüber hinaus noch verschiedene interne Zustände aufweisen kann, die dessen Verhalten beeinflussen, ist die theoretisch möglich Anzahl zu testenden Konstellationen möglicherweise noch sehr viel größer als bei ausschließlich validen Eingaben. Somit kann auch nur exemplarisches Testen bzgl. der Robustheit erfolgen.

Robustheitstests können stark von einer automatisierten Ableitung der Tests abhängen, da typischerweise sehr große (im Sinne vieler Daten), sehr komplexe (z. B. im Sinne von Vernetzungsgrad) oder sehr viele Tests die Basis bilden.

5.1.4 Ausschluss von Regressionen

Bei dem sogenannten *Regressionstesten*, vgl. z. B. [Lig02, S. 187 ff., Kap. 5.1.4], wird durch die Ausführung einer Testmenge nach einer Modifikation am System sichergestellt, dass mindestens die vorhandenen Testfälle auch nach der Änderung noch fehlerfrei „durchlaufen“. Dadurch sollen unerwünschte Seiteneffekte aufgrund bewusster und notwendiger Änderungen am System möglichst früh aufgedeckt und so ausgeschlossen werden. Regressionstests werden in der Praxis überwiegend automatisiert ausgeführt und ausgewertet.

5.1.5 Überprüfung nichtfunktionaler Eigenschaften

Neben den zuvor erwähnten Zielen, die sich – mit Ausnahme des abstrakten Vertrauens in eine Implementierung – alle durch objektive und technisch greifbare Kriterien bewerten lassen, existiert eine weitere Kategorie von Eigenschaften, die sich einer solchen Bewertung weitestgehend entziehen. Nichtfunktionale Eigenschaften eines *SUTs*, insbesondere solche, bei denen menschliche Empfindungen, Kultur und Ästhetik eine Rolle spielen, lassen sich schlecht algorithmisch untersuchen. Beispiele solcher Eigenschaften sind Benutzerakzeptanz, Look&Feel oder Benutzbarkeit (engl. Usability). Die Betrachtung konkreter Testtechniken zur Überprüfung solcher Eigenschaften, wie z. B. das Durchführen empirischer Studien, sprengen hier allerdings den Rahmen.

5.2 Konzepte und Terminologie

Im Folgenden werden die wesentlichen Konzepte und Begrifflichkeiten des Softwaretestens als Basis für tiefer gehende Betrachtungen eingeführt. Dabei muss allerdings bedacht werden, dass in der Literatur die Testterminologie nicht immer einheitlich und mit größtmöglicher Exaktheit definiert ist. Der folgende Abschnitt stellt das Ergebnis des Unterfangens dar, sich möglichst exakt an den gängigsten Definitionen zu orientieren. Die beiden wichtigsten Ressourcen, auf welche die im folgenden vorgestellte Testterminologie zurückzuführen ist, sind einerseits das standardisierte Glossar der IEEE für die Terminologie des Software-Engineerings [IEE90] und andererseits das Lehrbuch von P. Ammann und J. Offutt [AO08]. An Letzterem orientiert sich auch die in diesem Unterabschnitt gewählte Art der Darstellung. Eine weitere, häufiger zitierte Quelle für Definitionen rund um das Thema Testen von Software, ist der Standard zum Komponententesten der British Computer Society [Spe01].

5.2.1 Grundbegriffe des Testens

Nachdem in Definition 5.1 bereits formuliert wurde, was unter dem Vorgang des Testens zu verstehen ist, folgt nun die Definition eines einzelnen *Testfalls*.

Definition 5.2 (Testfall, vgl. [AO08, Def. 1.17, S. 15]):

Ein Testfall *(auch* Test-Case *oder einfach* Test*) ist ein Tupel* $T = (A, \Omega, I, E)$ *zusammengesetzt aus einer Menge* A *von Initialisierungsanweisungen, einer Menge* Ω *von Finalisierungsanweisungen, einer Menge* I *von Testeingaben, auch Stimuli genannt, und einer Menge* E *von erwarteten Ergebnissen. Je nach Art und Umfang des Tests können einzelne Mengen auch leer sein.*

Hierbei ist bemerkenswert, dass ein Test i. Allg. mehr umfasst als die Menge der Testeingaben. Zum einen sind Schritte notwendig, um das System in einen initialen Ausgangszustand zu überführen, zum anderen müssen Testausgaben nach dem eigentlichen Test erst aus dem System ausgelesen bzw. das System in einen definierten Zustand zurückgefahren werden. Darüber hinaus sind die erwarteten Ergebnisse bzw. erwarteten Eigenschaften der tatsächlichen Ergebnisse als Teile eines Tests aufzufassen. In den Definitionen 5.3 bis 5.6 werden die Testbestandteile einzeln eingeführt.

Definition 5.3 (Initialisierungsanweisung, vgl. [AO08, Def. 1.13, S. 15]):

Eine Initialisierungsanweisung *ist eine Eingabe für das zu testende System, die benötigt wird, um das System in einen definierten Ausgangszustand zu bringen, von dem aus das eigentlich zu testende Verhalten untersucht werden kann.*

Definition 5.4 (Finalisierungsanweisung, vgl. [AO08, Def. 1.14 - 1.16, S. 15]):

Eine Finalisierungsanweisung *ist eine Anweisung, die entweder benötigt wird, um das zu testende System nach dem eigentlichen Testen wieder in einen stabilen Zu-*

stand zu überführen, oder um eine Beobachtbarkeit, z. B. durch das Auslesen von Systemwerten, zu gewährleisten.

Definition 5.5 (Stimulus, vgl. [AO08, Def. 1.9, S. 14]):

Ein (Test-)Stimulus *oder eine* Testeingabe *ist eine benötigte Eingabe zur Ausführung des Systems, so dass das zu testende Systemverhalten unmittelbar ausgelöst bzw. entsprechend gesteuert und beeinflusst wird.*

Teststimuli können verschiedenste konkrete Ausprägungen besitzen. Oft handelt es sich um Methodenaufrufe inklusive Parameterbelegungen. Andere Optionen sind Signale, Literale (von primitiven Typen), Objekte, Referenzen oder Pointer, GUI-Aktionen, XML-Dokumente, Modelle im *MDSD*-Sinne, textuelle Artefakte (formuliert in einer formalen Sprache), Datensätze für oder aus einer Datenbank, Binärdaten etc.

Definition 5.6 (Erwartetes Ergebnis, vgl. [AO08, Def. 1.10, S. 14]):

Das erwartete Ergebnis *umfasst eine direkt oder indirekt beobacht- und überprüfbare Eigenheit des Systems, die sich während oder als Konsequenz der eigentlichen Ausführung einstellt. Nur wenn alle als obligatorisch angenommenen Ergebnisse nach dem Testen auch tatsächlich eingetreten sind, verhält sich das System wie erwartet, und damit fehlerfrei. Ansonsten wurde eine fehlerhafte Ausführung erkannt.*

Erwartete Ergebnisse können hierbei entweder in Form einer vollständigen Ausgabe vorgegeben sein oder mittels einer sog. *(partiellen) Orakelfunktion*, vgl. Definition 5.15, anhand (von Teilen) der Eingabe bestimmt werden. Für den ersten Fall kann beispielsweise auf spezifisches Domänenwissen oder auf Beispiele aus einer Spezifikation zurückgegriffen werden. Andere Orakel können auf viele verschiedene Arten realisiert werden. Eine offenkundige Option für ein Orakel wäre eine alternative Implementierung,[4] z. B. durch eine Vorgängerversion.

Für ein ausreichend gründliches Testen reicht ein einzelner Testfall i. d. R. nicht aus. Daher muss mit einer größeren Menge an Tests gearbeitet werden. Deshalb folgt in Definition 5.7 die Festlegung, wie Tests zu sogenannten Test-Suites zusammengefasst werden können.

Definition 5.7 (Testmenge, vgl. [AO08, Def. 1.18, S. 15]):

Eine Testmenge *(auch* Test-Suite*,* Test-Set*) ist eine nichtleere Menge M von* Testfällen*. Idealerweise sind die enthaltenen Tests voneinander vollkommen unabhängig (beispielsweise in beliebiger Reihenfolge oder sogar parallel aus-/durchführbar). Es kann aber auch eine Ordnungsrelation $R \subseteq M \times M$ gegeben sein, um beispielsweise eine Priorität oder inhärente Abhängigkeiten zwischen den Tests zu erfassen.*

[4] Man spricht dann häufig vom *Back-to-Back*-Testen, beispielsweise im Rahmen einer *N-Versionen-Realisierung*, vgl. z. B. [Lig09, S. 444].

5.2.2 Testziele, Anforderungen und Überdeckung

Umfang und Inhalt der Testmenge müssen zur *Intention des Testvorhabens* sowie sich daraus ergebenden objektiv *messbaren Anforderungen* passen. Beide Aspekte werden nun definiert.

Definition 5.8 (Testziel):

Als Testziel *bezeichnen wir hier die eigentliche Intention hinter dem Testvorhaben, also den konkret angestrebten exemplarischen Nachweis einer bestimmten (abstrakten) funktionalen oder nichtfunktionalen Eigenschaft des SUT.*

Das Ziel der meisten Testbemühungen ist, wie im vorangegangenen Abschnitt 5.1 erläutert, der experimentelle Nachweis der funktionalen *Korrektheit* (im Sinne der Anforderungen) und dadurch die Steigerung des Vertrauens. Dabei wird durch das Testen *bewiesen*, dass sich das *SUT für die gegebenen Tests* korrekt verhält und die Funktionalität tatsächlich realisiert wurde, zumindest für die getesteten Eingaben. Bei einer immer umfangreicheren Testmenge – eine möglichst systematische Erzeugung vorausgesetzt, die eine Verzerrung weitestgehend ausschließt – steigt die Wahrscheinlichkeit, dass eventuell vorhandene Fehler in einem beobachtbaren Effekt resultieren. Bleibt dieser aus, kann im Umkehrschluss angenommen werden, dass die Wahrscheinlichkeit für unentdeckte Fehler abnimmt. Allerdings ist eine Vergrößerung der Testmenge nicht unproblematisch. Einerseits bedingen größer werdende Testmengen einen gesteigerten Aufwand für die Erzeugung und Pflege der Tests sowie für die Ausführung auf dem *SUT*. Andererseits ist unklar, wann man mit dem Hinzufügen weiterer Tests aufhören sollte.

Aus den abstrakten Testzielen müssen also unmittelbar überprüfbare und konkrete Anforderungen an die Testmenge folgen. Solche Anforderungen lassen sich auf einer technisch-konzeptionellen Ebene in Form sog. *Testanforderungen* fassen, vgl. die nachfolgende Definition 5.9.

Definition 5.9 (Testanforderung, vgl. [AO08, Def. 1.20, S. 17]):

Eine Testanforderung *(engl. Test Requirement) ist eine Bedingung über der Testmenge bzw. deren Ausführung unter Bezugnahme auf Artefakte der Software. Erfüllt die Testmenge diese Bedingung nicht, so ist die Menge als unvollständig anzusehen.*

Testanforderungen werden mit dem Zweck formuliert, dass eine bestimmte Mindestqualität der Testmenge sichergestellt werden soll. Durch geeignete Wahl von Anforderungen kann so beispielsweise eine ausreichend große Vielseitigkeit der Tests gesichert werden. Ein Beispiel für eine konkrete Testanforderung wäre die Bedingung, dass ein bestimmtes Artefakt des *SUT* durch die Testmenge tatsächlich zur Ausführung gebracht wird. Man spricht dann davon, dass das Artefakt durch den Test *überdeckt* wurde.

Die Forderung, einen bestimmten Grad an Überdeckung bzw. eine bestimmte Überdeckung zu erreichen, also einen Schwellwert für das Verhältnis von erfüllten Testanforderungen zu vorhandenen Testanforderungen zu überschreiten, ist das weitverbreitetste Abbruchkriterium für das Hinzufügen weiterer Tests. Eine solche Forderung ergibt sich typischerweise aus einem sog. *Überdeckungskriterium* nach Definition 5.10, in das konkrete Implementierungen und ggf. empirische Erkenntnisse einfließen.

Definition 5.10 (Überdeckungskriterium, vgl. [AO08, Def. 1.21, S. 17]):

Ein Überdeckungskriterium *legt in Abhängigkeit zu den zugrunde liegenden Software-Artefakten fest, welche (Überdeckungs-)Anforderungen (im Sinne von Definition 5.9) – im Folgenden werden diese auch als* ***Coverage-Items*** *bezeichnet – für eine Testmenge bestehen, die bei Erfüllung die Testmenge als vollständig* ***ausreichend*** *qualifizieren. Die Menge der konkreten Anforderungen TR*[5] *ergibt sich durch systematisches Anwenden einer durch das Kriterium beschriebenen* ***Konstruktionsvorschrift*** *auf die Artefakte des SUT.*

Einige konkrete Beispiele für Überdeckungskriterien für das Testen klassischer Software werden weiter unten in Abschnitt 5.3 genannt. Der große Vorteil solcher Kriterien ist, dass sich die Erzeugung von Coverage-Items und die Überprüfung auf ihre Abdeckung durch Testwerkzeugen automatisieren lässt. Mit Hilfe empirischer Untersuchungen wurde für verschiedene Überdeckungskriterien untersucht, inwiefern sie Rückschlüsse auf die zu erwartende Fehlererkennungsleistung zulassen. Es wurden einige Studien publiziert, z. B. [FW93; Hut+94; And+06], die auf die Existenz entsprechender Zusammenhänge hindeuten, allerdings kann allgemein die Interpretation entsprechender Arbeiten schwierig und die Nützlichkeit und Übertragbarkeit ggf. limitiert sein, vgl. [IH14]. Dennoch ermöglichen die sich aus solchen Kriterien ergebenden Maßzahlen, vgl. Definition 5.11, Testmengen miteinander zu vergleichen. Damit lassen sich überflüssige Tests identifizieren oder zielgerichtet weitere Tests erzeugen.

Definition 5.11 (Überdeckungsmetrik):

Eine Überdeckungsmetrik*, auch* Überdeckungsmaß *genannt, ist eine mathematische Funktion f definiert über der Potenzmenge der nichtleeren Menge der abgeleiteten konkreten Überdeckungsanforderungen TR sowie der Kardinalität $\|TR\|$. Die Menge der abgeleiteten Überdeckungsanforderungen basiert auf einem bestimmten, zur Metrik gehörenden Überdeckungskriterium. Die Funktion bildet eine Teilmenge dieser Menge – die der erfüllten Kriterien – auf eine rationale Zahl x aus dem Intervall* $[0..1]$ *ab. Dies tut sie, indem sie ihr den Quotienten aus der Anzahlt der erfüllten zur Anzahl der vorhandenen Anforderungen zuordnet:*

$f : \mathscr{P}(TR) \times \mathbb{N}^* \to \{x \in \mathbb{Q} : 0 \leq x \leq 1\}\ ,\quad f(X,n)|_{TR} := \frac{\|X\|}{n}$*, mit*

$X \in \mathscr{P}(TR) \wedge X := \{t \in TR : t \text{ ist überdeckt}\}, n := \|TR\|.$

Dabei bedeutet ein Wert von 0, dass keine, und ein Wert von 1, dass alle Überdeckungsanforderungen erfüllt wurden.

Am Ende dieses Unterkapitels bleibt zu betonen, dass ein effektiver Testansatz neben technischen auch organisatorische Maßnahmen umfasst. Insbesondere die Erfassung und Dokumentation der Anforderungen sollte gewissenhaft erfolgen. Unter anderem diesem Zweck dient die sogenannte *Test-Spezifikation* nach Definition 5.12.

[5] Steht hier für *Test-Requirements*.

Definition 5.12 (Test-Spezifikation, vgl. [IEE08, Abschn. 3.1.49]):

Eine Test-Spezifikation *(auch* Test-Plan, Test-Beschreibung*) ist ein Dokument, das definiert* ***was*** *(die zu testenden Entitäten und die Granularität der Tests),* ***wie*** *(die eingesetzten Techniken, Werkzeuge und Testanforderungen, s. Definition 5.9),* ***wofür*** *(vgl. Definition 5.8), von* ***wem*** *(Rollen und Zuständigkeiten) und* ***wann*** *(Zeitpläne, Bezug auf Meilensteine etc.) getestet werden soll.*

5.2.3 Automatisierung und Werkzeuge

In der Praxis hängen Leistungsfähigkeit, Erfolg und Kosten eines Testverfahrens nicht nur von den reinen technischen Details ab. Auch die verfügbare Werkzeugunterstützung sowie der Automatisierungsgrad sind wesentlich. So sinkt beispielsweise der Aufwand für eine wiederholte Ausführung der Tests mit höherer Automatisierung. Dadurch kann die Akzeptanz für ein kontinuierliches Testen, z. B. bei Regressionstests, ansteigen. Auch können sowohl die Qualität als auch die Nachvollziehbarkeit des Testens dadurch profitieren, dass der Umfang an ermüdenden und unkreativen Arbeitsschritten, die manuell ausgeführt werden müssten, gesenkt wird. Damit wird es ebenfalls einfacher, zu protokollieren, welcher Aspekt mit welchem Resultat getestet wurde. Zusammengefasst bedeutet dies:

„To be effective and repeatable, testing must be automated“ [Bin99, S. 801]

Mit Hilfe sog. *Testgeneratoren*, s. die nachfolgende Definition 5.13, kann die Ableitung von Testfällen automatisiert werden, so dass sehr viele und/oder sehr große Testfälle möglichst praktikabel werden. Es existieren auch Verfahren, um speziell nach noch fehlenden Testfällen, beispielsweise bezogen auf ein Überdeckungskriterium, zu suchen, vgl. z. B. [LS05]. Ein weiterer positiver Aspekt von Testgeneratoren: sie sind, im Gegensatz zu Menschen, in der Lage, „unvoreingenommen“ zwischen Möglichkeiten auszuwählen und dadurch Stichprobenverzerrungen bei der Auswahl von Tests auszuschließen.

Definition 5.13 (Testgenerator):

Ein Testgenerator *ist ein Programm zur automatisierten und zielgerichteten Ableitung von Testfällen aus einer geeigneten formalen Problembeschreibung.*

Der Aufwand für die Konstruktion bzw. Anpassung eines Testgenerators ist allerdings nicht immer gerechtfertigt. Auch helfen generative Ansätze vor allem bei der initialen Erzeugung von Testmengen, also primär beim Einsparen von initialen Aufwendungen. Auch sollten menschlicher Verstand und Kreativität als Quelle für sinnvolle Testfälle keinesfalls unterschätzt und vernachlässigt werden. Diese „Gefahr“ besteht zumindest potentiell beim Einsatz von Testgeneratoren. Bereits die intellektuelle Auseinandersetzung mit dem SUT, z. B. mit dem Ziel der Überdeckung bestimmter Artefakte, kann bereits Qualitätsprobleme offenbaren, die kein automatisiertes Verfahren aufdecken könnte. Letztendlich hat die zugrunde gelegte Teststrategie nach Definition 5.14 erheblichen Einfluss auf den zu erwartenden Aufwand und die Qualität der Tests.

Definition 5.14 (Testgenerierungsstrategie):

Das geplante, schrittweise Vorgehen, welches ein Testgenerator (oder der menschliche Testentwickler) zum Erstellen der Testfälle an den Tag legt, wird im Folgenden als Testgenerierungsstrategie *oder kurz als* Teststrategie *bezeichnet.*

Eine Strategie gilt nach [How76] dann als **zuverlässig**, wenn sie stets zu einer endlichen Menge von Tests führt und letztere genau für den Fall, dass Fehler vorhanden sind, eben diese auch erkennen.[6]

Um entscheiden zu können, ob das Ergebnis des Tests positiv oder negativ ausfällt, muss in der überwiegenden Anzahl der Fälle die Ausgabe des *SUT* überprüft werden. Wie bereits weiter oben ausgeführt, definiert die Spezifikation des Problems das, was als korrekt anzusehen ist. Oft liegt es nahe, ein (komplettes) *erwartetes* Ergebnis anderweitig zu bestimmen (z. B. manuell oder mit Hilfe eines anderen Systems zu berechnen) und so vorzugeben, so dass dieses mit der tatsächlichen Ausgabe verglichen werden kann. Die Ableitung erwarteter Ergebnisse per Hand ist allerdings offenkundig kein praktikabler Ansatz, sobald die Testmenge eine bestimmte Größe oder Komplexität überschreitet. Darüber hinaus ist eine Testbewertung mittels Vergleich nur dann automatisierbar, wenn dazu die erwarteten Ergebnisse verfügbar sind – also entweder bereits vorliegen oder sich algorithmisch ableiten lassen – und ein ausreichend leistungsfähiger *Differencing (Diff)* Algorithmus zur Bestimmung der Unterschiede zur Verfügung steht. Allerdings gibt es auch noch andere Möglichkeiten, die Ausgabe eines *SUT* automatisiert zu bewerten, vgl. hierzu z. B. auch [AO08, Kap. 6.5, S. 230 ff.]. Die generische Bezeichnung einer solchen Funktionalität lautet *Testorakel*, vgl. hierzu auch noch mal Abbildung 1.1. Die nachfolgende Definition 5.15 legt diesen Begriff nun genauer fest.

Definition 5.15 (Orakel):

*Unter einem (Test-)*Orakel *verstehen wir zwei Dinge:*

(1) Eine Funktion oder ein Algorithmus, welche(r) aus einer Testeingabe wesentliche Eigenschaften einer zur Eingabe passenden korrekten Ausgabe oder diese selbst ableiten kann, ***ohne*** *dabei selbst eine (vollständige) Implementierung (für alle möglichen Eingaben) der zu entwickelnden bzw. der zu testenden Funktionalität sein zu müssen. Die tatsächlichen Ausgaben des SUT bzw. dessen Eigenschaften können anschließend mit den „Vorhersagen“ des Orakels verglichen werden (manuell oder automatisiert).*

(2) Eine Softwarekomponente, die eine Orakelfunktion nach Teil (1) der Definition umfasst oder deren Ausgabe unmittelbar nutzt, um damit den eigentlichen Vergleich zwischen erwartetem und tatsächlichem Ergebnis durchzuführen und so das Ergebnis der Bewertung produziert.

Im weiteren Verlauf der Arbeit werden einige problemangepasste Orakeloptionen aus der Literatur aufgegriffen und beschrieben, weshalb hier auf eine Auflistung weitere Optionen verzichtet wird. Für eine Übersicht verschiedener technischer Realisierungsmöglichkeiten für Orakel, die *nicht* auf vorberechnete Ergebnisse oder eine alternative Im-

[6] Howden konnte zeigen, dass eine effektive Strategie, die für *beliebige* Testaufgaben zuverlässig ist, nicht existiert.

plementierung angewiesen sind, sei hier auf einen Technischen Bericht von Baresi und Young verwiesen [BY01]. Zumindest im Kontext des Testens von *MTs* werden sog. *partielle* (engl. partial) und *vollständige* (engl. full) Orakel-Funktionen unterschieden. Erstere überprüfen nur einige isolierte, als wichtig angenommene Eigenschaften der Ausgabe, letztere basieren dagegen auf einem vollständigen Modellvergleich zwischen Ausgabe und vorberechneten Ergebnissen, vgl. z. B. [Sch+13].[7]

Für das Ergebnis der Bewertung einer Testausgabe durch das Orakel hat sich eine eigene Bezeichnung etabliert. Sie wird hier in Definition 5.16 formell eingeführt.

Definition 5.16 (Verdikt):

Das Endergebnis eines Orakels wird als Verdikt *bezeichnet. Es umfasst die Aussage, ob der Test* ***erfolgreich*** *war (die Software also weiterhin als fehlerfrei gilt), der Test als* ***fehlgeschlagen*** *gilt (die Software also unerwartetes oder offenkundig fehlerhaftes Verhalten aufweist) oder der Testlauf mit* ***undefiniertem*** *Ergebnis abgebrochen wurde. Letzteres Resultat kann, je nach Bedarf, einer der beiden anderen Möglichkeiten zuordnet werden.*

In ihrer Gesamtheit bilden die zum Testen eingesetzten Werkzeuge das sog. *Testrahmenwerk* nach Definition 5.17 (vgl. hierzu auch noch mal die Abbildung 1.1). Entsprechende Werkzeuge sind für verschiedene Sprachen verfügbar; eine auch nur in Ansätzen vollständige Übersicht würde den Rahmen dieser Arbeit sprengen.[8] Hier soll nur die Eclipse-*IDE* (in der Version für die Java-Entwicklung) als eine Möglichkeit für eine technische Basis erwähnt werden. Sie enthält mit ihrer *JUnit*[9]- sowie *Ant*[10]-Unterstützung bereits einige zentrale Komponenten für ein Testrahmenwerk. Darüber hinaus lassen sich eventuell fehlende Funktionen durch Plugins nachrüsten – naheliegende Optionen wären beispielsweise (i) zur Messung der Überdeckung (z. B. *JaCoCo*,[11] *EclEmma*[12] oder *Cobertura*[13]), (ii) Werkzeuge zum GUI-Testen (z. B. *Jubula*[14]) oder (iii) zur Generierung von Testreports etc. (z. B. *TestNG*[15]).

Definition 5.17 (Testrahmenwerk):

Ein Testrahmenwerk *(synonym zu* Testumgebung*,* Test-Harness *oder* Test-Driver*) umfasst die Infrastruktur zum interaktiven oder automatisierten Testen. Potentielle*

[7] In [MBL08] wurde das Adjektiv „partiell“ im Zusammenhang mit Zusicherungen verwendet. Auf wen die erstmalige Verwendung beider Begriffe in diesem Zusammenhang zurückgeht, konnte von mir nicht abschließend in Erfahrung gebracht werden.

[8] Die englischsprachige Wikipedia listete am 14. Januar 2014 beispielsweise Unit-Testing-Frameworks für über 70 verschiedene Sprachen auf: `http://en.wikipedia.org/w/index.php?title=List_of_unit_testing_frameworks&oldid=590684908`

[9] Zurzeit bekanntestes Unit-Testing-Framework für Java, `http://junit.org/` (abgerufen am 14.1.2014).

[10] Build-Files auf Basis von *XML* und Java, `http://ant.apache.org/` (abgerufen am 14.1.2014).

[11] Java Code Coverage Library, `http://www.eclemma.org/jacoco/` (abgerufen am 14.1.2014).

[12] `http://www.eclemma.org/` (abgerufen am 14.1.2014).

[13] `http://cobertura.github.io/cobertura/` (abgerufen am 14.1.2014).

[14] `http://www.eclipse.org/jubula/` (abgerufen am 14.1.2014).

[15] `http://testng.org/` (abgerufen am 14.1.2014).

Bestandteile wären Laufzeitumgebungen, Bibliotheken, Repositories (für Tests oder benötigte Daten), Test- und Reportgeneratoren, Testorakel etc. aber auch Skriptsprachen und andere Möglichkeiten zur automatisierten Durchführung, Aus- und Bewertung der Tests.

5.3 Überdeckungskriterien

Nachdem in Abschnitt 5.2.2 mit den Definitionen 5.10 und 5.11 bereits die beiden zentralen Begriffe der Testüberdeckung eingeführt wurden, sollte auch deutlich geworden sein, dass Überdeckung bzw. Coverage im Wesentlichen als „Maß der Vollständigkeit der Tests“ [Bei90, S. 14] dient. Dieser Abschnitt ist der Vorstellung der wichtigsten konkreten Techniken und Strategien zur Ableitung von Coverage-Items vor allem für textuelle und strukturierte Programmiersprachen gewidmet. Die betrachteten Kriterien lassen sich auf „klassische“ Software, geschrieben in einer der derzeit populären imperativen Programmiersprachen (Java, C/C++, etc.), unmittelbar anwenden. Auf Programmier- und Transformationssprachen, die den üblichen Paradigmen und Konventionen weniger bis gar nicht folgen, lassen sie sich nur zum Teil übertragen.

Für Überdeckungskriterien existieren zwei typische Anwendungsfälle, s. [AO08, S. 18]: Zum einen ermöglichen entsprechende Kriterien die *Bewertung einer Testmenge* im Hinblick auf die durch sie erzielte Überdeckung. Zum anderen ermöglichen die Kriterien die *zielgerichtete Suche* nach bzw. das zielgerichtete Erstellen von bisher noch fehlenden Tests.

Die Darstellung der Überdeckungskriterien erfolgt hier in Anlehnung an die von Ammann und Offutt in [AO08] verwendete Klassifikation. Die Kriterien sind folglich in vier Klassen unterteilt, die sich anhand der Art des zugrunde liegenden Artefakts bzw. dessen Struktur ergeben. Dabei weisen die Überdeckungskriterien nach [AO08] einen Bezug zu (i) Graphen, (ii) Logikausdrücken, (iii) Partitionen oder (iv) Syntaxbeschreibungen auf. Vor der Betrachtung konkreter Kriterien dieser vier Klassen – in der Reihenfolge der obigen Nennung – wird zuerst noch auf den nicht unwichtigen Aspekt der allgemeinen *Vergleichbarkeit von Kriterien* eingegangen.

5.3.1 Zusammenhänge zwischen Kriterien

Die Entscheidung für oder gegen ein Überdeckungskriterium, bei einer Auswahl aus einer Menge möglicher Kriterien, wirft die Frage nach der grundsätzlichen Vergleichbarkeit von Kriterien auf. Eine Möglichkeit besteht darin, zu untersuchen, ob eine Testmenge, die den Anforderungen eines der Kriterien genügt, auch immer die Anforderungen eines anderen Kriteriums erfüllt. Sollte dies der Fall sein, so wäre das erste Kriterium mindestens so streng und das Testen intuitiv somit mindestens so „gründlich“,[16] wie bei Verwendung des zweiten Kriteriums. Als Vorgriff auf die konkreten Kriterien sein darauf hingewiesen, dass dies in der Praxis tatsächlich vorkommt und auch abschließend gezeigt werden kann. Ein solcher Zusammenhang zwischen Kriterien wird als *Subsumption*, vgl. Definition 5.18, bezeichnet.

[16] Ammann und Offutt betonen in [AO08, S. 21] allerdings, dass es für einen allgemeingültigen positiven Effekt beim Testen mit strengeren Kriterien, wenn überhaupt, nur schwache (empirische) Belege gibt.

Definition 5.18 (Subsumption, vgl. [AO08, Def. 1.24, S. 19]):

Als Subsumption *(im Deutschen oft auch* Subsumtion*) bezeichnet man eine Ordnungsrelation*[a] *'$\succcurlyeq$' definiert über Paaren (c_1, c_2) von Überdeckungskriterien. Mit CC sei hier die Menge aller betrachteten Kriterien bezeichnet. Dabei bedeutet $c_1 \succcurlyeq c_2$ (kompakt für $(c_1, c_2) \in \succcurlyeq$, mit $\succcurlyeq \subseteq CC^2$), dass, sobald Kriterium c_1 erfüllt ist, daraus folgt, dass auch Kriterium c_2 erfüllt ist.*

[a] Genauer gesagt, handelt es sich um eine Quasiordnung (engl. Preorder).

Aus der fehlenden Forderung nach Antisymmetrie in Definition 5.18 ergibt sich, dass aus $c_1 \succcurlyeq c_2 \wedge c_1 \preccurlyeq c_2$ nicht gefolgert werden kann, dass $c_1 = c_2$. Folglich ist es möglich, dass unterschiedliche Kriterien zu vergleichbaren bzw. äquivalenten Forderungen führen.

Überdeckungskriterien lassen sich darüber hinaus selbstverständlich auch anhand weniger formaler Eigenschaften vergleichen. So gibt es Überdeckungskriterien, die für das betrachtete *SUT* zu besonders vielen oder zu besonders schwer zu erfüllenden Anforderungen führen. Ihr Einsatz kann also höhere Kosten verursachen bei ggf. unklarem Nutzen gegenüber einfacheren Kriterien. Wie in [AO08, Abschn. 1.3.2, S 20 f.] nahegelegt, ist es vorteilhaft, für die Testerzeugung möglichst leicht und direkt zu erfüllende Kriterien haben, die dennoch zu hinreichend guten Tests führen. Bei der Bewertung von Testmengen sollten dagegen besser bewertete Mengen auch tatsächlich leistungsfähiger (i. S. v. mehr Fehler aufdecken) Testmengen identifizieren. Auch sind zielführende Hinweise auf ungetestete Aspekte des *SUT* gefragt.

5.3.2 Graphbasierte Kriterien

Die erste und bekannteste Gruppe der Überdeckungskriterien sind die graphbasierten Kriterien, vgl. [AO08, Kap. 2.2, S. 27 ff.]. Sie basieren auf der Erkenntnis, dass sich anhand der Struktur einer Implementierung oder anderer Design- oder Entwurfsartefakte einfache (gerichtete) Graphen konstruieren lassen, die i. d. R. isomorph zu den Strukturen sind, auf denen sie basieren. Die auftretenden Graphen sind i. d. R. *weder* zyklenfrei noch frei von Mehrfachkanten. Man denke beispielsweise an die Graphstruktur einfacher Zustandsautomaten. Knoten und/oder Kanten der Graphen können, falls benötigt, darüber hinaus mit Labels annotiert sein. Speziell ausgezeichnete initiale und finale Knoten sind möglich. Ein konkretes Überdeckungskriterium kann sich dann, grob gesagt, entweder alleine auf die Bestandteil des Graphen – also seine Knoten und/oder Kanten bzw. dessen *Struktur* – beziehen oder auf die Annotationen der Elemente. Dann aber auch im Zusammenspiel mit der Struktur des Graphen. Im ersten Fall spricht man von *strukturellen* oder, etwas einschränkend, von *kontrollflussorientierten* Kriterien. Im zweiten Fall dagegen von *datenflussorientierten* Kriterien. In beiden Fällen werden die ableitbaren Coverage-Items mittels Pfaden in den Graphen abgedeckt. Letztere ergeben sich anhand der Eigenschaften konkreter Testläufe als Traversierung des Graphen.

Strukturelle Überdeckung

Bei der strukturellen Überdeckung liegt das Ziel darin, die unmittelbaren Bestandteile des Graphen bei einer Traversierung im Rahmen der Testläufe zu erreichen und damit zu

überdecken. Das einfachste Überdeckungskriterium der strukturellen Überdeckung verlangt, dass jeder erreichbare Knoten im Zuge der *Node-Coverage*, s. [AO08, S. 33], durch Tests überdeckt wird. Es muss folglich für jeden Knoten mindestens einen Ausführungsablauf durch den Graphen geben, der diesen erreicht. Eine analoge Forderung bzgl. der Überdeckung einzelner Kanten ist als *Edge-Coverage*, s. [AO08, S. 34], bekannt. Darüber hinaus gehende Forderungen über die Abdeckung von Kanten-Tupeln sind möglich (und sinnvoll), vgl. z. B. die „Transition-Pair Coverage“ aus [Off+03].

Erweiterte strukturelle Überdeckungskriterien beziehen sich auf Forderungen bzgl. der Abdeckung konkret vorgegebener oder anderweitig charakterisierter *Pfade*. Das strengste Kriterium beinhaltet die Forderung, *alle* möglichen Pfade zu überdecken. Häufig ist allerdings die Zahl der möglichen Pfade sehr groß, so dass eine Berücksichtigung aller als nicht sinnvoll erscheint. Enthält der Graph darüber hinaus Zyklen bzw. Schleifen, sind sogar beliebig viele Pfade möglich und ein praktikables Überdeckungskriterium muss sich auf eine endliche Teilmenge beschränken. Bzgl. zyklischer Pfade kann beispielsweise auch die Überdeckung dieser bis zu einer Obergrenze für die Zyklendurchläufe gefordert werden. Oder es wird gefordert, dass für jeden Knoten genau ein zyklischer Pfad überdeckt wird, der bei diesem beginnt und endet, vgl.z. B. die in [AO08, S. 36] vorgestellte *Round-Trip-Coverage*.

Repräsentiert der Testgraph den Kontrollflussgraphen eines imperativen oder strukturierten Programms, so steht die Knotenüberdeckung für die sog. *Anweisungs-* bzw. *Blocküberdeckung* (auch als *Statement-Coverage* oder *Basic-Block-Coverage* bezeichnet, vgl. [AO08, S. 54] aber auch Abbildung 5.1). Kantenüberdeckung würde im Falle eines Kontrollflussgraphen dagegen der sog. *Zweigüberdeckung* entsprechen. Die Pfadüberdeckung wird im Kontext von Kontrollflussgraphen ebenfalls als solche bezeichnet, s. den Knoten „Pfadüberdeckungstest“ in Abbildung 5.1. Sie verlangt somit, dass alle oder bestimmte Anweisungs- bzw. Blocksequenzen durch Tests durchlaufen werden.

Entstammt der Testgraph dagegen beispielsweise einem Zustandsautomaten, vgl. hierzu beispielsweise [Kim+99], so repräsentiert die Forderung nach Knotenüberdeckung tatsächlich die Forderung nach *Zustandsüberdeckung*. Entsprechend steht die ursprüngliche Kantenüberdeckung für die Forderung nach *Transitionsüberdeckung*. Laut Ammann und Offutt geht die Idee der strukturellen Graphüberdeckung sogar ursprünglich auf frühe Arbeiten zum Testen von *Finite State Machines (FSM)* wie [Cho78] zurück, s. [AO08, Abschn. 2.8, S. 100].

Datenflussbasierte Überdeckung

Die datenflussbasierte Überdeckung nutzt ebenfalls die Struktur von Graphen, vgl. [AO08, Abschn. 2.2.2, S. 44 ff.] bzw. [RW85; FW88]. Allerdings wird durch die dedizierte Berücksichtigung von schreibenden und lesenden Zugriffen auf Variablen sichergestellt, dass neben dem Kontrollfluss auch der Informations- bzw. Datenfluss durch die unterschiedliche Variablennutzung bei der Testauswahl bzw. Testerstellung Berücksichtigung findet.

Dazu werden die Knoten und Kanten des Graphen mit Informationen darüber annotiert, welche Variablen in einem dem Graphelement zugrunde liegenden Artefakt durch Zuweisung *definiert* (sog. „*def*“-Verwendung der Variable) oder *gelesen* (sog. „*use*“-Verwendung der Variable) werden. Für den „*use*“-Fall können noch Zugriffe aufgrund von Berechnungen („*computation-use*“, „*c-use*“) von Zugriffen bei der Auswertung von Bedingungen zur Verzweigung („*predicate-use*“, „*p-use*“) unterschieden werden, vgl. [RW85].

Die Kernidee der Überdeckung besteht nun darin, dass sog. *def-use*-Paare (*DU*-Paare), die sich jeweils auf einzelne Variablen beziehen, durch Tests abzudecken sind. Das bedeutet, dass mindestens ein Testfall dafür sorgen muss, dass das Graphelement, welches die geforderte *def*-Verwendung der betrachteten Variablen umfasst, erreicht wird. Anschließend muss durch besagten Testfall ein Pfad durch den Graphen genommen werden, der unbedingt frei von weiteren *def*-Verwendungen der betrachteten Variable ist, und der schließlich die eingeforderte *use*-Verwendung erreicht. Der große Vorteil gegenüber rein strukturellen Überdeckungskriterien liegt dabei darin, dass bei der Datenflussüberdeckung nur eine endliche Menge von Pfaden traversiert werden muss.

Wie streng die Anforderungen an die Testmenge tatsächlich sind, lässt sich sowohl anhand der eingeforderten *DU*-Paare (z. B. *All-Def*-Coverage[17] vs. *All-Use*-Coverage[18]) als auch anhand der zu überdeckenden Pfade zwischen diesen (z. B. ein beliebiger Pfad bei *All-Use*-Coverage vs. *alle* möglichen Pfade bei *All-DU-Path*-Coverage) kontrollieren, vgl. ebenfalls [RW85]. Die datenflussbasierte Überdeckung kann beispielsweise auf Kontrollflussgraphen von imperativen oder strukturierten Programme angewendet werden [FW88]. Aber auch eine Verwendung bei (*UML*-)Zustandsautomaten bzw. erweiterten Endlichen Automaten ist möglich [Kim+99].

5.3.3 Logikbasierte Kriterien

Eine weitere Möglichkeit zur Definition von Überdeckungskriterien bieten Ausdrücke einer zweiwertigen Logik, vgl. [AO08, Kap. 3, S. 104 ff.]. Diese treten regelmäßig in Softwareartefakten wie Programmen, Spezifikationen und Automaten auf (z. B. als Schleifen- oder Verzweigungsbedingungen, Transitionswächter, etc.). Konkrete Überdeckungskriterien fordern in diesem Zusammenhang, dass bestimmte (Teil-)Ausdrücke im Rahmen des Testens zu jeweils beiden möglichen Logikwerten ausgewertet werden. Auch sind kombinierte Vorgaben für mehrere (Teil-)Ausdrücke gleichzeitig möglich.

Die Grundlage bildet die *Aussagenlogik* (engl. Propositional Logic) mit ihren beiden Wahrheitswerten (`wahr` bzw. '$\top$' und `falsch` bzw. '$\bot$'), atomaren Aussagen, denen durch Auswertung ein Wahrheitswert zugeordnet werden kann, sowie logischen Verknüpfungsoperatoren und auch Klammerungen. Durch Verknüpfung ist es möglich, komplexere, zusammengesetzte Aussagen zu bilden. Aussagenlogische Ausdrücke lassen sich in Verbindung mit den Verknüpfungen $\neg$ (Negation $\triangleq$ *NICHT*), $\wedge$ (Konjunktion $\triangleq$ *UND*), $\vee$ (Disjunktion $\triangleq$ *ODER*), auch als (zweiwertige) *Boolesche Algebra* interpretieren und entsprechend manipulieren und umformen. Für weitere Details sei an dieser Stelle auf entsprechende Grundlagenliteratur verwiesen, z. B. [HR04].

Aussagen als unveränderlich und atomar anzunehmen, stellt sich als zu unflexibel heraus, um damit Überdeckungskriterien formulieren zu können. Sollen beispielsweise Verzweigungspunkte in einem Programmablauf – die Entscheidung über die Verzweigung erfolgt meist aufgrund veränderlicher Variablenwerte – durch Logikausdrücke beschrieben werden, so stellt man fest, dass der Wahrheitsgehalt der (Teil-)Aussagen von den dynamischen Umständen abhängt. Um solche „kontextabhängigen" Aussagen formulieren zu können, werden *Prädikate*, wie in Definition 5.19 beschrieben, benötigt.

[17] Überdeckung mind. eines *DU*-Pfades für jede *def*-Verwendung (*use*-Verwendung beliebige gewählt).
[18] Überdeckung mind. eines Pfades zwischen jedem möglichen *DU*-Paar.

Definition 5.19 (Prädikat, allg.):

Ein Prädikat $P(.)$ *ist eine durch (nicht Boolesche) Variablen parametrisierte Aussage, die noch nicht (vollständig) auswertbar ist. Die konkrete Auswertung eines Prädikats setzt voraus, dass alle Variablen* ***gebunden*** *bzw. deren Werte bestimmt sind*[a]*; aus dem Prädikat entsteht dann eine Aussage. Letztere ist dann (aufgrund ihrer Interpretation) entweder* `wahr` *oder* `falsch`*.*
Die Anzahl der freien Variablen eines Prädikats bestimmt dessen ***Stelligkeit****, wobei 0-stellige Prädikate möglich sind und atomaren Aussagen entsprechen.*

[a] Durch Einsetzen konkreter Werte (oder per Quantifizierung mittels ∃- und ∀-Quantoren).

Definition 5.19 ist allerdings noch zu unspezifisch und allgemein für die Formulierung konkreter Überdeckungskriterien, weshalb sie durch die Definitionen 5.20 und 5.21 ergänzt wird.

Definition 5.20 (Prädikat, speziell (nach [AO08, S. 104 f.])):

Prädikate *umfassen*
(i) boolesche Variablen,
(ii) Ausdrücke über nicht-booleschen Variablen und Literalen, die mittels Verknüpfungen durch übliche Vergleichsoperatoren[a] *gebildet werden,*
(iii) Funktionsaufrufe, die Variablen als Parameter entgegennehmen können,
sowie Verknüpfungen dieser über die üblichen logischen Operatoren/Junktoren $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\oplus$, $\leftrightarrow$*. Letztes ist möglich, da alle Ausdrücke der Form (i-iii) auf* `wahr` *oder* `falsch` *abgebildet werden können.*

[a] $\{>, <, \geq, \leq, =, \neq\}$

Auf Basis der Prädikate aus Definition 5.20 lässt sich bereits ein praktisch nutzbares Überdeckungskriterium, die sog. *Prädikatabdeckung*, formulieren: Für jedes Prädikat P aus einer vorgegebenen Menge von Prädikaten, die zum Testen berücksichtigt werden, muss ein Testfall existieren, der dazu führt, dass P während der Ausführung einmal zu '$\top$' und einmal zu '$\bot$' evaluiert (vgl. [AO08, Kriterium 3.12, S. 106]). Legt man nun die jeweils *vollständigen* logischen Bedingungen bei der Steuerung des Kontrollflusses (bei `if-`, `for-`, `while`-Anweisungen) als Prädikate fest, so führt das Kriterium bereits dazu, dass alle Zweige im Kontrollfluss traversiert werden. Es subsumiert dann die Zweigüberdeckung (die Zweigüberdeckung subsumiert in dieser Konstellation auch die Prädikatüberdeckung). Aufgrund dieser Ähnlichkeit werden Logikkriterien häufiger auch den kontrollflussorientierten Verfahren zugerechnet, vgl. Abbildung 5.1.

Da sich allerdings auch Fehler in der Formulierung der logischen Bedingungen ergeben können, die sich nur unter gewissen Umständen auf die Gesamtentscheidung durchschlagen, ist es sinnvoll, auch Teilaussagen zu überdecken. Dazu wird das Konzept einer sog. Klausel benötigt, s. Definition 5.21.

Definition 5.21 (Klausel, vgl. [AO08, S. 105]):

Ausdrücke der Form (i-iii) aus Definition 5.20 enthalten selbst keine logischen Operatoren. Solche „logisch atomaren" Teilausdrücke, also Prädikate, die keine logischen Junktoren enthalten, werden Klauseln *genannt.*

Wählt man die zu überdeckenden Prädikate im Falle der Verzweigungsbedingungen nun so aus, dass sie ausschließlich den elementaren Bedingungen der Verzweigungs- und Schleifenbefehle entsprechen – es werden also nur die Klauseln der zuvor betrachteten Prädikate berücksichtigt – dann ergeben sich die Anforderungen der sog. *Klauselüberdeckung* [AO08, S. 106]. Folglich wird für jede Klausel gefordert, dass sie im Rahmen des Testens einmal zu '$\top$' und einmal zu '$\bot$' evaluiert. Bezogen auf die Entscheidungspunkte im Programmcode nennt man diese einfache Form der Klauselüberdeckung auch *einfache Bedingungsüberdeckung* (engl. Simple Condition Coverage) [Lig02]. Die einfache Bedingungsüberdeckung subsumiert die Zweigüberdeckung i. Allg. *nicht*.

Fordert man, dass für einzelne (größtmögliche) Prädikate ihre Klauseln zu allen möglichen Kombinationen an Wahrheitswerten evaluieren, so fordert man die sog. (vollständige) *Mehrfach-Bedingungsüberdeckung* (engl. Multiple Condition Coverage), vgl. [Lig02]. Das Kriterium subsumiert die Prädikatüberdeckung sowie alle weniger vollständigen Formen der Klauselüberdeckung (*Bedingungs-/Entscheidungsüberdeckung*, *minimale Mehrfachbedingungsüberdeckung*, *Modified Condition Decision Coverage (MCDC)*).

5.3.4 Partitionsbasierte Kriterien

Eine weitere Möglichkeit zur Formulierung von Überdeckungskriterien basiert auf der Analyse der Definitionsmengen der Eingabedaten analog zu [OB88], vgl. auch [AO08, Kap. 4, S. 150-169]. Möchte man beispielsweise eine Methode in Java testen, so kann man davon ausgehen, dass diese oft Eingabeparameter besitzt. Unter der Annahme, dass die Selbstreferenz `this` immer als eine Art impliziter Parameter mit übergeben wird, kann man sogar verallgemeinern, dass immer mindestens ein Parameter existiert. Als weitere implizite Parameter bzw. Eingabedaten der Operation können auch noch der Zustand des Objektes oder aber des gesamten *SUT* gesehen werden sein.

Die Hauptaufgabe des partitionsbasierten Testens liegt nun darin, ein sog. *Input Domain Model* [AO08, S. 152] zu entwickeln, indem (i) jeder relevante Parameter bzw. allgemeiner alle relevanten (abstrakten) Charakteristika der Eingabe identifiziert, (ii) die theoretisch möglichen Definitionsmengen der einzelnen Parameterwerte bzw. Charakteristikausprägungen bestimmt und (iii) diese jeweils vollständig in (überlappungsfreie) Partitionen zerlegt werden, um dann anschließend eine Überdeckung anhand der Auswahl und Kombinierbarkeit von Partitionen zu definieren, vgl. [AO08, Abschn. 4.1, S. 152 ff.]. Aus Sicht des Testens werden dabei alle konkreten Werte innerhalb einer Partition, also die konkreten Repräsentanten dieser, als „gleich gut" geeignet bzw. äquivalent angenommen. Dem liegt die Prämisse zugrunde, dass ein Test mit einem beliebigen Repräsentanten weder besser noch schlechter als ein Test mit jedem anderen der möglichen Werte ist. Ein (abstrakter) Testfall ist nach [AO08] ein Tupel der ausgewählten Partitionen, wobei für jeden Parameter bzw. jede Charakteristik genau ein der jeweils möglichen Partitionen auszuwählen ist.

Zur Bestimmung der Partitionen bieten sich nach [AO08] zwei wesentliche Ansätze:

1. Syntaxorientierte Partitionierung anhand der Parameter auf Grundlage von Methodensignatur bzw. Schnittstellenbeschreibung.
2. Semantische Partitionierung anhand funktionaler Äquivalenzen bei der Eingaben. Dies kann verschiedene Kombinationen für Parameterwerte voraussetzen.

Entsprechende Partitionen sollten entweder anhand von Methodensignaturen oder Spezifikationen bzw. Zusicherungen abgeleitet werden. Spezielle („magische“) Werte, ungültige Eingaben, Intervallgrenzen und Extremwerte eignen sich ggf. auch, um als eigene Partitionen berücksichtigt zu werden. Ggf. können Details der Implementierungsstruktur im Zuge von sog. *Pfadbereichstests*, vgl. [Lig09, Abschn. 5.2.2], genutzt werden, um geeignete Domänen zu bestimmen, wie beispielsweise in [WC80] vorgeschlagen.

Die eigentlichen Überdeckungskriterien repräsentieren Strategien zur Auswahl und Kombination der Partitionen. Man spricht deswegen auch vom *kombinatorischen Testen*, vgl. z. B. [GOA05]. Betrachtet man die Parameter jeweils isoliert, so sollte ein Repräsentant aus jeder identifizierten Partition in der Testmenge vorhanden sein. Da sich manche Fehler aber nur unter gewissen Eingabekonstellationen zeigen, die dann von mehr als einem Wert abhängen, sollten auch die möglichen Kombinationen von Partitionen systematisch abgedeckt werden. Alle *möglichen* Kombinationen abzudecken erscheint nur auf den ersten Blick sinnvoll (da sehr gründlich), ist aber praktisch oft nicht realisierbar.[19] Diese beiden Erkenntnisse liefern die Motivation für selektive Strategien, wie die *Pair-Wise*- oder die *T-Wise*-Abdeckung. Für einen umfassenden Überblick hierzu inkl. Hinweisen zu leistungsfähigen Heuristiken s. [GOA05].

5.3.5 Syntaxbasierte Kriterien

Bei syntaxbasierten Überdeckungskriterien wird die syntaktische Struktur von (meist textuellen) Artefakten für das Testen ausgenutzt. Es gibt nach [AO08] zwei wesentliche Ausprägungen, die sich allerdings deutlich voneinander unterscheiden: (i) Überdeckungskriterien für formale Grammatiken (klassischerweise BNF- bzw. *EBNF*-basiert; Graphgrammatiken sind, wie im nachfolgenden Kapitel 6 beschrieben, hierfür aber ebenfalls geeignet) und (ii) die sog. Mutationsüberdeckung. Hier wird nur kurz auf Punkt (i) eingegangen, da dem Mutationstesten aufgrund seiner Bedeutung für diese Arbeit der eigene Abschnitt 5.4 gewidmet ist.

Syntaxbasiertes Testen auf Grundlage von Grammatikbeschreibungen wird genutzt, um beispielsweise textuelle Eingabedaten mit einer bestimmten Syntax zu generieren. Dies ist unter anderem dann sinnvoll, wenn Parser [Pur72] oder Compiler getestet werden sollen, vgl. z. B. [DH81]. Eine Testüberdeckung kann hierbei über der Anwendung einzelner Produktionen der Grammatik (auch „Rule-Coverage“), s. [Pur72], oder aber anhand von Abhängigkeiten zwischen diesen (z. B. die „Context-Dependent Branch Coverage“ nach [Läm01]) definiert werden. In [AO08, S. 172] werden des Weiteren noch die sog. *Terminal Symbol Coverage* sowie die *Derivation Coverage* definiert. Erstere ergibt sich als Überdeckung aller Terminalsymbole, letztere als impraktikable Überdeckung aller ableitbarer Strings der beschriebenen Sprache.

[19] Die Anzahl der Kombinationen wächst näherungsweise exponentiell mit der Parameteranzahl n als Exponent. Der genaue Wert ergibt sich zu $\prod_{i=1}^{n} |P_i|$, s. [GOA05], wobei $|P_i|$ die Anzahl der Partitionen für den i-ten Parameter (bzw. Charakteristik) ist.

5.3.6 Herausforderungen

Allen Überdeckungskriterien gemeinsam sind einige wenige grundlegende Herausforderungen, die sich bei ihrem Einsatz ergeben, insbesondere in folgenden Zusammenhängen:

1. **Nichterfüllbarkeit** (engl. Infeasibility) – Bei der Ableitung konkreter Coverage-Items kann es vorkommen, dass Eigenschaften verlangt bzw. Anforderungen aufgestellt werden, die durch keine Testmenge für das konkrete *SUT* erfüllt werden können. Beispielsweise können einzelne abzudeckende Pfade in einem Kontrollflussgraphen aufgrund impliziter Abhängigkeiten zwischen Entscheidungen ausgeschlossen sein. Soll als Testziel eine hundertprozentige Abdeckung erreicht werden, müssen solche Forderungen von der Betrachtung ausgeschlossen werden.

2. **Abdeckung als Stoppkriterium** – Um auf Grundlage der ableitbaren Coverage-Items ein Stoppkriterium für das Testunterfangen zu etablieren, wird ein Schwellwert für die Überdeckungsmetrik benötigt. Vollständige Abdeckung ist oft nur unter sehr großen Anstrengungen oder gar nicht zu erreichen. In der Praxis wird oft ein niedrigerer Abdeckungsgrad gefordert, wobei offen bleiben muss, welcher Wert sinnvoll ist. Auch ist ein globaler Wert über alle Artefakte hinweg nicht immer günstig.

3. **Mapping-Problem** (vgl. [AO08, S. 137]) – Manche Überdeckungskriterien führen zu Testanforderungen, aus denen sich nur ein Teil der wesentlichen Eigenschaften konkreter Tests ergibt. Die resultierenden abstrakten Tests müssen, um ausführbar zu sein, somit noch in konkrete Testeingaben übersetzt werden. Als Beispiel hierfür sei auf funktionsbasierte Charakteristika beim partitionsbasiertem Testen verwiesen, die zwar festlegen, was einen guten Test ausmacht, aber zum Teil offen lassen, wie dies auf konkrete Eingabewerte abgebildet werden soll.

4. **Testgenerierung** – Eine Testmenge nach einem konkreten Kriterium zu bewerten, ist, die entsprechende Werkzeugunterstützung vorausgesetzt, ein relativ einfacher Schritt. Deutlich schwieriger ist es dagegen, fehlende Tests zu ergänzen. Folglich ist es wichtig, dass das Kriterium brauchbare Hinweise dafür liefert, wie fehlende Tests beschaffen sein müssen. Ein passender Generierungsansatz auf Grundlage eines Kriteriums und des zu testenden Artefakts ist aus Benutzersicht wünschenswert, allerdings stellt dies auch eine Herausforderung dar, die nicht immer sinnvoll realisierbar ist.

5.4 Mutationstesten

Mutationstesten, auch *mutationsbasiertes Testen* oder *Mutationen-Testen*, ist ein Verfahren, bei dem ein *SUT* (i. d. R. in Form eines Programms; ausführbare Spezifikationen sind aber auch geeignet [JH11]) oder eine Testmenge mit Hilfe einer Menge aus dem zu testenden Artefakt abgeleiteter Varianten, den sog. *Mutanten*, vgl. Definition 5.22, im Rahmen einer dynamischen Ausführung und anschließenden Bewertung der Ausgaben untersucht wird, [DLS78; How82; DeM+88]. Eine grundlegende Einführung bieten unter anderem [AO08, Kap. 5] oder [Lig09, Abschn. 5.1.3, S 185 ff.]. Für einen sehr umfangreichen und gut aufbereiteten Überblick über den Themenkomplex siehe auch [JH11].

Mutationstesten kann dabei auf zwei Arten interpretiert werden:

(i) Als Prozess zur zielgerichteten Konstruktion einer *effektiven und diversifizierende*[20] Testmenge in dem Sinne, dass für jeden vorhandenen Mutanten mindestens ein Testfall existieren muss, der diesen erkennt. In diesem Zusammenhang ist dann auch von *Mutation-Coverage* [AO08, S. 175] bzw. *Mutation-Score* nach Definition 5.26 die Rede. Als Orakelfunktion dient bei dieser Verwendungsart i. d. R. der direkte Vergleich zwischen dem *SUT* und dem gerade betrachteten Mutanten. Die Fehlersensitivität der so abgeleiteten Tests wird maßgeblich durch die Auswahl und die Beschaffenheit der Mutanten und damit durch die Wahl der sog. *Mutationsoperatoren* nach Definition 5.23 bestimmt, vgl. hierzu auch [AO08, S. 173 f.].

(ii) Als Ansatz zur Bewertung einer gegebenen Testmenge und, darauf aufbauend, zur Bewertung von Testverfahren, die zur Ableitung von Tests geeignet sind, vgl. [Lig02, Kap. 5.1]. Hierbei ist die eingesetzte Orakelfunktion nicht zwingend als vergleichsbasiert anzunehmen, da die Tests auf andersartigen Orakelfunktionen basieren können. Wichtig ist, dass die eingeführten Defekte auch zu beobachtbar fehlerhaftem Verhalten führen und dies durch das genutzte Orakel erkennbar ist. Kann das unmutierte System als korrekt angesehen werden und ist die Ausgabe eindeutig bzw. der Diff-Algorithmus in der Lage, äquivalente Lösungen sicher zu erkennen, so stellt der Vergleich mit diesem ein optimales Orakel dar.

Definition 5.22 (Mutant, vgl. [AO08, Def. 5.47, S. 173]):

Ein Mutant *ist eine Variante des zugrunde liegenden Programms, die mit diesem* ***fast*** *komplett übereinstimmt. Der einzige Unterschied besteht in einer einzelnen, kleinen, lokal begrenzten und an der Syntax orientierten Modifikation. Dabei ist die Modifikation das Resultat der einmaligen und isolierten Anwendung des* Mutationsoperators *(nach Definition 5.23) auf das ursprüngliche Programm.*

Das *SUT* und ein davon im Sinne der Definition 5.22 abgeleiteter Mutant unterscheiden sich folglich unmittelbar syntaktisch voneinander. Allerdings bedingt dies häufig auch einen semantischen Unterschied, da Mutationsoperatoren einerseits typischerweise realistische Fehler von Entwicklern emulieren sollen und außerdem zu beobachtbar unterschiedlichem Verhalten führen sollen. Es kann nicht garantiert werden, dass letzteres tatsächlich eintritt; algorithmisch ist die Frage nach Äquivalenz bzw. Nichtäquivalenz zweier Programmvarianten i. Allg. nicht entscheidbar, wie diesbezüglich in [OP96; BA82] bemerkt. Es existieren allerdings Verfahren, mit denen äquivalente Mutanten in *bestimmten Situationen* identifiziert werden können [OC94].

Um Mutanten systematisch und automatisiert erzeugen zu können, benötigt man Umsetzungen des in Definition 5.23 eingeführten *Mutationsoperator*-Konzepts, welche die systematischen Modifikationen am eigentlichen *SUT* beschreiben, vgl. hierzu z. B. auch [AO08, Def. 5.46, S. 173]. Um unterschiedliche Arten von Modifikationen zu unterscheiden, lassen sich einzelne *spezifische Mutatoren* definieren – wie in Definition 5.24 geschehen – die durch einen eindeutigen Namen gekennzeichnet sind und nur eine einzelne Strategie zur Modifikation umsetzen.

[20] Der Begriff „*diversifizierend*“ bedeutet hier, dass das ursprüngliche System und sein Mutant hinsichtlich ihres Verhaltens verglichen werden, mit dem Ziel, dass der Mutant durch abweichendes Verhalten erkannt wird. Der Begriff selbst wurde [Lig02, Abschn. 5.1] entnommen.

Definition 5.23 (Mutationsoperator, vgl. [AO08, Def. 5.46, S. 173]):

Ein Mutationsoperator *ist ein Algorithmus* $\mathtt{mut}$*, der durch Eingabe eines Programms P einer unterstützten Sprache, einer Anwendungsstelle x sowie einer Modifikationsart m parametrisiert ist. Bei seiner Ausführung verändert er P auf die durch m vorgegebene Art und Weise an der Anwendungsstelle x minimal ab. Es entsteht der Mutant P' als Variante von P, der anschließend ausgegeben wird:* $P' \leftarrow \mathtt{mut}(P, x, m)$.

Wie die Anwendungsstelle des Mutationsoperators ausgewählt wird, ist in Definition 5.23 noch offen gelassen. Eine naheliegende Option besteht allerdings darin, einzelne Anwendungsstellen zufällig aus der Menge aller möglichen Anwendungsstellen auszuwählen, falls nicht alle möglichen Mutanten erzeugt werden sollen.

Definition 5.24 (Spezifischer Mutator):

Sei $f(m)$ eine Funktion, die jeder Modifikation $m \in M$ aus der Menge M der betrachteten Mutationsarten einen eindeutigen Namen[a] $\$n \in \Sigma^*$ *zuweist:* $f(m) : M \to \Sigma^*$. *Wir vereinbaren, dass wir einen mit einer beliebigen, aber festen Mutationsart m_0 parametrisierten Mutationsoperator mittels dessen eindeutigem Namen* $\$n = f(m_0)$ *referenzieren können. Des Weiteren benutzen wir die vereinfachende Schreibweise* $\$n(P, x)$ *für* $\mathtt{mut}(P, x, m_0)$*, wenn wir den so definierten* spezifischen Mutator $\$n(.)$ *meinen.*

[a] Ein String über dem Zeichenalphabet Σ.

Mutatoren sind in der Vergangenheit für diverse Sprachen entwickelt worden; [JH11] nennt beispielsweise Arbeiten zu FORTRAN, Ada, C, Java, SQL und C#. Für einen Überblick zu Mutationswerkzeugen sei auf [JH11, Tabelle 7] verwiesen.

Je nach Art der Modifikation, der Programmgröße sowie des Ziels des Mutationstestens kann es sinnvoll sein, dass alle theoretisch möglichen Mutanten für ein Programm zu generieren. Für diese erschöpfende Strategie wird in Definition 5.25 der Begriff des *erweiterten Mutationsoperators* eingeführt.

Definition 5.25 (Erweiterter Mutationsoperator):

Ein erweiterter Mutationsoperator *ist ein Algorithmus* $\mathtt{mut}^*$*, der für die Eingabe eines Programms P und einer Modifikation m* alle *Anwendungsstellen $X(P, m)$ bestimmt, und für jede Anwendungsstelle $x \in X$ einen Mutanten $P'_{x,m}$ mittels des einfachen Mutationsoperators* $\mathtt{mut}$ *ableitet, so dass sich als Ausgabe die Menge aller entsprechenden Mutanten P'^* ergibt:*
$P'^* \leftarrow \mathtt{mut}^*(P, m)$*, mit* $\mathtt{mut}^*(P, m) = \bigcup_{\forall x \in X(P,m)} \{\mathtt{mut}(P, x, m)\} = \bigcup_{\forall x \in X(P,m)} \{P'_{x,m}\}$.

Analog zu Definition 5.24 ließen sich auf Basis von Definition 5.25 erweiterte spezifische Mutatoren definieren, die jeweils alle Mutanten eines bestimmten Typs für alle Anwendungsstellen erzeugen. Auf eine entsprechende Definition wird hier verzichtet.

Nachdem nun bekannt ist, wie Mutanten erzeugt werden, stellt sich die Frage, wie sie konkret verwendet werden. Dazu muss das Konzept der *Mutationsanalyse* betrachtet werden. Gemeint ist damit die Verwendung der Mutantenmenge entweder zur (i) Einführung

einer weiteren Form der Testüberdeckung, vgl. die nachfolgende Definition 5.26, oder zur (ii) Bewertung einer Testmenge hinsichtlich ihrer Adäquatheit nach Definition 5.27 bzw. Definition 5.28.

Definition 5.26 (Mutationsüberdeckung, vgl. [AO08, S. 175]):

Die Mutationsüberdeckung *(auch* Mutation-Score*) ist der Quotient aus der Anzahl der durch Tests identifizierten*[a] *Mutanten gegenüber der Gesamtzahl an abgeleiteten Mutanten (i. d. R. abzüglich der als äquivalent erkannten Mutanten).*

[a] man spricht dabei auch von „getöteten“ Mutanten

Nach [AO08, S. 178] kann, je nach Ableitungsstrategie, die Mutationsüberdeckung andere Überdeckungskriterien simulieren – z. B. Anweisungsüberdeckung durch Ersetzen einzelner Anweisungen durch Laufzeitfehler auslösende Substitute – oder Tests für typische Codierungsfehler einfordern. Zum Erkennen eines Mutanten muss sich dessen Verhalten beobachtbar vom ursprünglichen Programm bzw. dem fehlerfreien System unterscheiden. Je nachdem, ob sich die Beobachtbarkeit auf das Ausgabeverhalten des *SUT* oder auch auf interne Zustände bezieht, z. B. ausgelöst durch das Verhalten von mutierten Komponenten, unterscheidet man zwischen *starker* und *schwacher* Mutationsüberdeckung [How82].

Definition 5.27 (Adäquatheit, vgl. [DLS78]):

Eine Testmenge ist adäquat*, falls sie in der Lage ist, Fehler zuverlässig aufzudecken.*

Da die Adäquatheit einer i. d. R. endlichen und nichterschöpfenden Testmenge für beliebige Programme i. Allg. nicht nachweisbar ist, beziehen sich praktisch nutzbare Adäquatheitsanforderungen auf konkrete Überdeckungskriterien und schränken so die Anforderungen auf unmittelbar überprüfbare Maße ein. Insbesondere die in Definition 5.28 beschriebene Mutationsadäquatheit ist von Interesse, da man durch sie – zwei wesentliche Annahmen vorausgesetzt – eine ausreichend realistische Abschätzung der Leistungsfähigkeit einer Testmenge erhalten kann, vgl. [DLS78], falls die Mutationen halbwegs sinnvoll gewählt wurden. Bei den Annahmen handelt es sich einerseits um den als kompetent erachteten Entwickler, der fehlerfreie Programme anstrebt und von diesen, wenn überhaupt, dann nur geringfügig abweicht. Und andererseits um den sog. *Coupling-Effect*, vgl. hierzu ebenfalls [DLS78], der besagt, dass eine Testmenge, die mehrere einfache Fehler zu finden in der Lage ist, auch komplexere Fehler mit hoher Wahrscheinlichkeit aufdecken kann.

Definition 5.28 (Mutationsadäquatheit, vgl. [AO08, S. 181]):

Eine Testmenge ist mutationsadäquat *hinsichtlich einer abgeleiteten Menge von Mutanten, falls sie in der Lage ist, alle nichtäquivalenten Mutanten des SUT zu identifizieren.*

Die Leistungsfähigkeit des mutationsbasierten Testens wurde, wie beispielsweise in [Off+96], durch empirische Untersuchungen gezeigt und mit der anderen Überdeckungskriterien verglichen. Darüber hinaus spricht eine grundsätzliche Automatisierbarkeit für das Verfahren. Nachteilig wirken sich dagegen die Notwendigkeit von speziell angepassten, sprachspezifischen Mutatoren, der hohe technische Aufwand sowie Probleme aufgrund äquivalenter Mutanten aus. Auch große Laufzeitkosten durch die n-fache Testausführung sowie die teilweise Neugenerierung von Mutanten nach Korrekturen am *SUT* können praktische Herausforderungen darstellen.

5.5 Modellbasiertes Testen

Modellbasiertes Testen respektive *MBT*, vgl. [Dal+99; Cho78], ist ein eher allgemeiner Begriff und umfasst, je nach Interpretation, eine ganze Reihe konkreter Techniken, wie z. B. anhand der Übersichtsarbeiten von Utting et al. [UPL12] oder Dias Neto et al. [Dia+07] zu sehen ist. Einen guten Überblick über den Themenkomplex bietet [Lig09, Kap. 6, S. 215 ff.] bzw. das Standardwerk [UL07] von Utting et al., an der sich die folgenden Ausführungen orientieren. Für eine (formalere) Betrachtung der Thematik im Zusammenspiel mit reaktiven Systemen s. auch [Bro+05].

Nach [UL07, S. 7] sind verschiedene Interpretationen des *MBT*-Begriffs möglich:

1. Ableitung von Testeingaben aus einem *Domänenmodell* (z. B. aus einem Klassifikationsbaum).
2. Ableitung von Testfällen anhand eines *Umgebungsmodells*, das die Umgebung des *SUT* (z. B. anhand sog. Markow-Kette) beschreibt.
3. Ableitung von Testfällen anhand eines ausführbaren *(System-)Modells* (also eines Verhaltensmodells; das Modell beschreibt bereits das zu testende Systemverhalten ganz oder teilweise und kann so auch als Orakel dienen).
4. Die Aufgabe des Ableitens konkreter Testskripte aus *abstrakten Beschreibungen* (z. B. Übersetzen eines Sequenz-Diagramms in eine Serie konkreter API-Aufrufe).

Verbreitet sind vor allem die Interpretationen eins und drei. Im weiteren Verlauf sind vor allem Ansätze auf Basis eines Verhaltensmodells gemeint, also grob das, was Liggesmeyer in [Lig09, S. 215] als „modellbasierte Testtechniken im engeren Sinne“ bezeichnet. In der Praxis umfassen i. d. R. alle *MBT*-Ansätze auch Teile des vierten Punkts, da die abgeleiteten Tests an das zu testende System adaptiert werden müssen. Man unterscheidet diesbezüglich zwischen *abstrakten* und *konkreten* Tests, vgl. die nachfolgenden Definitionen 5.29 und 5.30.

Definition 5.29 (Abstrakter Test, vgl. [UL07, S. 9]):

Ein Test, dessen Eingaben (und ggf. erwartete Ausgaben) unter Bezugnahme auf das zum Testen verwendete Modell *formuliert sind, und der hierdurch nicht auf konkrete Sprachmittel, APIs, Funktionen oder Entitäten des eigentlichen SUTs angewiesen ist, wird als* abstrakter Test *bezeichnet.*

Ein Beispiel für einen abstrakten Test nach Definition 5.29, wäre eine Sequenz von Ereignissen zur Stimulation eines Zustandsautomaten, so dass ein vorgegebener Ablauf durch den Automaten erfolgt. Das eigentliche *SUT* entspricht normalerweise nicht dem Testmodell, so dass die Tests noch für das *SUT konkretisiert* werden müssen.

Definition 5.30 (Konkreter Test, vgl. [UL07, S. 9 u. S. 29]):

Einen vollständigen, implementierungs- und systemspezifischen und auf dem SUT unmittelbar ausführbaren Test, bezeichnet man als konkreten Test*. Er entspricht einem der möglichen Ergebnisse der Adaption (mindestens) eines abstrakten Tests an das konkrete SUT.*

Allen zuvor genannten Interpretationen ist gemeinsam, dass die Tests nicht anhand der eigentlichen Implementierung oder einer der Implementierung zugrunde liegenden Spezifikation ableitet werden. Stattdessen werden die Tests anhand eines speziell entwickelten Testmodells erzeugt, welches im Idealfall nur die für das Testen relevanten Informationen umfasst. Übergeordnetes Ziel ist stets, dass die Tests möglichst automatisiert (teil-)generiert werden können. Dies stellt den wesentlichen Unterschied zum *klassischen* Black-Box-Testen dar, wie das folgende Zitat unterstreicht:

„Model-based testing is the automation of the design of black-box tests.“ [UL07, S. 8]

Wichtig ist es auch festzuhalten, dass es für den Einsatz von *MBT*-Verfahren unerheblich ist, ob auch die Entwicklung des *SUT* modellbasiert/-getrieben erfolgt.

Für den Einsatz von *MBT*-Verfahren kann sprechen, dass (i) Implementierungsdetails des Systems unbekannt sind, da das System noch nicht existiert oder aber entsprechende Entwicklungsartefakte nicht verfügbar sind, (ii) sich durch das Modell nennenswerte Vereinfachungen durch eine Reduktion auf das wesentliche Systemverhalten ergibt, (iii) besonders viele (generierte) Tests benötigt werden oder (iv) ein *geeignetes* Modell[21] bereits vorliegt. Darüber hinaus bieten Modelle aufgrund Ihrer potentiell hohen Abstraktion die Möglichkeit zum bewussten Zulassen von *Nicht-Determinismus*. Möglich ist, eine nicht vorhersagbare Auswahl zwischen mehreren alternativen Ausführungen oder nicht-beobachtbaren Zustandsübergängen im *SUT* zu modellieren, vgl. hierzu z. B. [Bro+05, Def. 22.3, S. 615]. Dadurch müssen Design-Entscheidungen, wie sie im Rahmen einer Implementierung zu treffen sind, noch nicht vorweggenommen werden.

Ein in [UL07, Abschn. 2.2, S. 26 ff.] vorgestellter generischer Prozess zum Einsatz von *MBT*-Techniken umfasst die folgenden Arbeitsschritte: (i) Modellierung, (ii) Generierung, (iii) „Konkretisierung“ der abstrakten Tests, (iv) Ausführung und (v) Analyse der Ergebnisse. Die Erzeugung der benötigten Modelle im ersten Schritt sollte dabei weitestgehend unabhängig von bereits vorhandenen Implementierungsartefakten erfolgen, damit eventuell darin vorhandene Fehler erkannt werden können und keine „blinden Flecken“ entstehen. Entscheidend für Qualität und Umfang der erzeugten Testmenge ist dabei der zweite Schritt im Prozess, da hierfür eine Auswahl der zu generierenden Tests erfolgen muss. Dies setzt voraus, dass ein entsprechendes Auswahlkriterium für eine Automatisierung zur Verfügung steht, vgl. [UL07, S. 28]. Auswahlkriterien solcher Gestalt basieren in der Regel auf einem geeigneten, möglicherweise problemspezifischen Überdeckungsbegriff, welcher wiederum über den eingesetzten Modellarten selbst definiert ist. Als Beispiele sei hier auf die bereits auf S. 93 erwähnten Konzepte der *Zustands-* oder *Transitionsüberdeckung* bei endlichen Automaten verwiesen.

[21] Ein solches Modell sollte *funktionsspezifische Details* umfassen und *technische Implementierungsdetails* aussparen. Einem *MDSD*-Ansatz steht dies nicht grundsätzlich entgegen. Allerdings läuft ein solcher Ansatz aber einer erwünschten Unabhängigkeit zwischen Test- und Implementierungsmodell zuwider (vgl. [Bro+05, Abschn. 2.3]).

[...] Wenn ich weitergeblickt habe, so deshalb, weil ich auf den Schultern von Riesen stehe.

(Isaac Newton, aus einem Brief an Robert Hooke, nach [Wes96, S. 143])

6 Stand der Forschung

In den vorangegangenen Abschnitten wurden bisher die wesentlichen Grundbegriffe und Konzepte vorgestellt, die für das Verständnis und die Verortung der weiteren Teile der Arbeit notwendig sind. Den Übergang zu den Kernbeiträgen der Arbeit stellt nun dieses Kapitel dar. Es fasst die wichtigsten verwandten Arbeiten und Ansätze zum Testen und weiteren (formalen) Verifikationstechniken mit Bezug zur *MDSD* sowie zu *MTs* überblicksartig zusammen.

Das Kapitel ist in drei Abschnitte unterteilt. Im ersten Abschnitt wird, als Ergänzung zu Abschnitt 5.5, kurz auf Arbeiten aus dem Bereich des *modellbasierten Testens* eingegangen. Der Bezug zur Arbeit ist deshalb gegeben, weil (a) Testverfahren für *MTs* grundsätzlich im Zuge von modellbasierten Testverfahren eingesetzt werden können und (b) einige Testansätze für *MTs* wiederum auf Verfahren des modellbasierten Testens zurückgreifen. Danach folgt eine Betrachtung von Arbeiten zur Qualitätssicherung von *UML*-Modellen mit einem Schwerpunkt auf Klassen- und Aktivitätsdiagrammen. Der dritte Abschnitt widmet sich schließlich der Qualitätssicherung von Modelltransformationen, hauptsächlich durch testende Verfahren, aber auch im Sinne der formalen Verifikation und auch der Validierung.

6.1 Modellbasiertes Testen

Wie bereits in Abschnitt 5.5 dargelegt, verbergen sich hinter dem *MBT*-Begriff unterschiedliche konkrete Techniken. Dies ist vor allem der verschiedenartigen Modellarten geschuldet, die zum Einsatz kommen. Die nachfolgende (unvollständige) Liste nennt einige der eingesetzten Modellarten, die in der Literatur als Grundlage zum Testen vorgeschlagen werden:

- (erweiterte) endliche Automaten bzw. *FSMs*, s. [Cho78; LY96] (vgl. auch [Lai02] für entsprechende Möglichkeiten beim Testen von Kommunikationsprotokollen),

- Statecharts wie UML-Zustandsautomaten, vgl. z. B. [Kim+99; Wei10],
- Petri-Netze (auf Basis sog. *Unfoldings*), wie z. B. in [UK97], bzw. allgemein Transitionssysteme (mit potentiell unendlichen oder überabzählbaren Zustands-/Transitionsmengen),
- Beschreibungen in speziellen (textuellen) Spezifikationssprachen, wie z. B. der *Java Modeling Language (JML)* [Bur+05], der *OCL* [Ali+11] oder der Z-Notation [CM09],
- Sequenzdiagramme und *MSCs*, vgl. [Bin99, Kap. 8.5],
- Markow-Ketten, s. z. B. [WT94],
- Graphtransformationsregeln, vgl. z. B. [HM05; KA07; KRH12b; Gue+13],
- (Simulink-)Blockschaltbilder,[1] vgl. z. B. [SPK12].

Utting et al. unterscheiden in ihrer Klassifikation [UPL12] (unter Bezugnahme auf [Lam00]), vgl. auch [UL07, S. 374, Abb. 11.1], insgesamt sieben Modellparadigmen, auf die sich die zum Testen geeigneten Modellarten verteilen. Einige der dort aufgeführten konkreten Beispiele sind hier in den Klammern mit angegeben:

- *Snapshot*[2]*-basierte* Notationen mit Vor- und Nachbedingungen für Operationen (z. B. Z-Spezifikationen),
- *Transitionsbasierte* Notationen mit Zuständen (z. B. *FSMs*),
- Notationen auf Basis *zeitlicher Verläufe* (z. B. Temporale Logiken, *MSCs*),
- Notationen auf Basis *mathematischer Modelle* (z. B. sogenannte Algebraische Spezifikationen[3] wie in [GMH81; BGM91]),
- *Operationale* Notationen (z. B. Prozessalgebren, Petri-Netze, *HDLs*),
- Notationen zur Beschreibung *stochastischer Prozesse* (z. B. Markow-Ketten),
- *Datenflussnotationen* (z. B. *Lustre* [Hal+91] oder Blockdiagrammsprachen wie z. B. *Simulink*).

Ein weiterer wichtiger Aspekt, auf den [UPL12] eingeht, und einer der größten Gewinne durch den Einsatz von *MBT*-Techniken, besteht in der automatisierbaren Testerzeugung. Entsprechende Basistechnologien sind nach [UPL12], ohne dabei auf die jeweiligen Ansätze im Details einzugehen: (i) *Model-Checking* (vgl. z. B. [EFM97]), (ii) *symbolische Ausführung* (vgl. z. B. [Cla76; Kin76] oder [Cla+02] auf Basis von [RBJ00]) sowie (iii) Verfahren, die auf (semi-)automatischen Beweisen für Theoreme (vgl. z. B. [BW13]) aufbauen. Darüber hinaus lassen sich *MBT*-Verfahren auch mit dem mutationsbasierten Testen kombinieren, wie beispielsweise in [ABM98] beschrieben.

Abschließend sei an dieser Stelle für eine Übersicht zu den diversen *MBT*-Ansätzen auch noch einmal auf die Arbeiten von Dias Neto et al. [Dia+07] sowie von Utting und Legeard [UL07] verwiesen.

6.2 Qualitätssicherung bei UML-Modellen

Für das *modellbasierte Testen* von Implementierungen auf *der Grundlage von UML-Diagrammen* gelten grundsätzlich die Aussagen zum *MBT* aus den Abschnitten 5.5 und 6.1. Beispielsweise lassen sich auf Grundlage von *UML*-Klassendiagrammen im Zusam-

[1] Vorzugsweise für Hardware-Tests, im Kontext sog. *HiL*-Ansätze.
[2] Der Systemzustand wird durch eine Menge von Variablen und deren Wertebelegungen beschrieben.
[3] Diese Art der Spezifikation ist nicht gleichzusetzen mit Algebraischen Graphtransformationen [Ehr+06].

menspiel mit Zustandsautomaten auch Tests generieren; ein entsprechender Ansatz wurde z. B. in [SMF99] vorgestellt. Die Tests werden hierbei mit Hilfe eines *KI*-basierten Planungssystems abgeleitet.

Auch die *OMG* hat sich als treibende Kraft hinter der *UML* dem *MBT*-Thema gewidmet und mit dem *UML Testing Profile (UTP)* einen einschlägigen Standard verabschiedet, vgl. [13b]. Ein Ziel der Standardisierung liegt darin, „Spezifikationen für einen *MBT*-Ansatz methoden- und domänenneutral und unabhängig von einem konkreten Typsystem“ zu erstellen, s. [13b, S. 1]. Dazu werden entsprechende Konzepte definiert und auch eine Abbildung auf *JUnit* beschrieben.

In der Praxis werden sehr häufig Zustandsautomaten zur Spezifikation oder auch als Implementierungsmodell eingesetzt. Entsprechende Überdeckungskriterien für *UML*-Statecharts wurden beispielsweise in [OA99] auf Grundlage sogenannter `Change`-Events[4] vorgestellt. Ursprünglich wurde dieser Ansatz als Basis für das modellbasierte Testen der Implementierung eines Zustandsautomaten entwickelt, ist aber auch auf unmittelbar ausführbare Statecharts anwendbar. Der Ansatz nutzt aus, dass Booleschen Ausdrücke (im Zusammenspiel mit dem aktuellen Zustand) steuern, ob und wann die mit einem `Change`-Event verbundenen Aktionen tatsächlich auslösen. Auf Grundlage der Ausdrücke können dann, wie von Offutt und Abdurazik beschrieben, verschiedene Überdeckungsarten definiert und auch entsprechende Tests generiert[5] werden. Konkret werden die folgenden Überdeckungskriterien eingeführt:

1. Transitionsüberdeckung,
2. Prädikatüberdeckung,
3. Überdeckung von Paaren angrenzender Transitionen,
4. Überdeckung von (vor-)gegebenen Transitionssequenzen.

Eine weitere Arbeit mit ähnlicher Ausrichtung ist [BCL03]. Briand et al. gehen darin allerdings von weniger starken Einschränkungen aus als die Autoren in [OA99]. So werden beispielsweise zusätzlich Event-Typen neben `Change`-Events zugelassen, beispielsweise sogenannte `Call`-Events (inklusive Parameter). Auch werden die Auswirkungen schaltender Transitionen aufgrund zugehöriger Aktionen auf den Systemzustand berücksichtigt und *OCL*-Ausdrücke zur Formulierung von Wächterausdrücken, Vor- und Nachbedingungen zugelassen. Briand et al. beschreiben außerdem, wie unter gewissen Voraussetzungen sog. *Testsequenzen* automatisch bestimmt werden können, aus denen dann teilautomatisch komplette Tests ableitbar sind.

Für den weiteren Verlauf dieser Arbeit sind *UML*-Zustandsautomaten von nachrangigem Interesse. Sie wurden hier allerdings nicht ohne Grund berücksichtigt, da die bei *SDM* zur Steuerung der *GT*-Regeln verwendete und an Aktivitätsdiagramme angelehnte Kontrollflusssprache eine gewisse syntaktische Ähnlichkeit zu Zustandsautomaten aufweist. Dies lässt sich auf die ursprüngliche Sichtweise von Aktivitätsdiagrammen als spezielle Zustandsautomaten im Rahmen der ersten Versionen des *UML*-Standards zurückführen (vgl. hierzu [03b, S. „2–170“]). Grundsätzlich ließen sich also die zuvor erwähnten Überdeckungskriterien aus [OA99] auch auf die Kontrollflussanteile von Story-

[4] In der konkreten Syntax durch das Schlüsselwort `when` gekennzeichnet.
[5] Mit den Werkzeugen UmlTest bzw. SpecTest [Off+03].

und *SDM*-Diagrammen übertragen. Dies entspräche, in Übereinstimmung mit den Aussagen zu den strukturellen graphbasierten Kriterien aus Abschnitt 5.3.2, den Kanten- und Pfadüberdeckungskriterien für den Kontrollflussgraphen.

Speziell für *UML*-Diagramme an sich (bzw. Software-Spezifikationen auf Basis der *UML*) existieren auch angepasste Überdeckungs- und Adäquatheitsbegriffe für das Testen, z. B. in den Arbeiten [MP05] respektive [Gho+03]. In [Din+05] wurde dagegen ein angepasstes Fehlermodell als Grundlage für die Verifikation sowohl von *UML*-basierten Design- als auch von Implementierungsmodellen vorgestellt. Da sie von größerer Bedeutung für diese Arbeit sind, werden im Folgenden Testansätze für Klassen- und Aktivitätsdiagramme dediziert behandelt.

6.2.1 Testen von Klassendiagrammen

Klassendiagramme sind zweifelsohne eine sehr wichtige, wenn nicht gar die wichtigste Teilsprache der *UML*, vgl. [Unh05, S. 10, Tabelle 1.2]. Dies wird auch intuitiv deutlich, wenn man sich vor Augen führt, dass die offizielle *UML*-Spezifikation diverse Klassendiagramme umfasst und nicht nur die anderen Teilsprachen mit ihrer Hilfe modelliert werden. Klassendiagramme der *UML* respektive der *MOF* bilden darüber hinaus die Grundlage der *Metamodellierung*. Somit wird deutlich, dass Klassendiagramme als zentrale Entwicklungsartefakte eines *MDSD*-Projektes einer gründlichen Qualitätssicherung unterzogen werden sollten.

Testen stellt dabei nur eine der Möglichkeiten zur Verifikation (und Validierung) dar. In der Praxis werden Klassendiagramme bzw. Metamodelle aufgrund ihrer statischen Natur eher selten getestet. Viel verbreiteter ist die Übersetzung in einen geeigneten Formalismus, wie ihn einige der im Folgenden aufgegriffenen Quellen beschreiben, falls beispielsweise die Erfüllbarkeit (i. S. v. Instanziierbarkeit) bzw. Widerspruchsfreiheit von Nebenbedingungen in Form von *OCL*- oder Multiplizitätsvorgaben gezeigt werden soll. Mit [And+03] von Andrews et al. existiert allerdings zumindest eine einschlägige Arbeit, die sich im Speziellen mit der Überdeckung von Elementen eines Klassendiagramms (bzw. Metamodells) und im Allgemeinen mit der Überdeckung von *UML*-Designmodellen im Rahmen des Testens beschäftigt. Diese Arbeit hat darüber hinausgehend unmittelbare Relevanz für einige Arbeiten, die sich mit dem Testen von Modelltransformationen beschäftigen, weshalb sie hier genauer betrachtet werden soll.

In besagter Arbeit [And+03] wird ein Ansatz zum Testen von Software-Designs, welche als Menge von *UML*-Diagrammen für eine nachgelagerte Implementierung vorliegen, vorgestellt. Dazu werden einerseits Klassendiagramme betrachtet, da sie „Objektkonfigurationen zum Testen“ [And+03] beschreiben. Andererseits werden Diagrammarten zur Spezifikation von Interaktionen (konkret: Kollaborationsdiagramme) berücksichtigt, da sie sinnvolle „zu testende Sequenzen von Nachrichten“ [And+03] beschreiben. Die für Klassendiagramme vorgeschlagenen Überdeckungskriterien sind im Einzelnen:

1. ***AEM*** (Association-End Multiplicity) – Für jede Assoziation des betrachteten Klassendiagramms müssen in der Menge der Tests – die Testeingaben bestehen aus Objektdiagrammen – Objekte und Links dergestalt existieren, dass jede „repräsentative“ Paarung der potentiell möglichen Multiplizitäten (für die Enden der Assoziation) als eine der zu überdeckenden Situationen, durch ein entsprechendes Objekt-Link-Geflecht abgedeckt wird.

Existiert beispielsweise eine Assoziation A zwischen den Klassen C_1 und C_2 mit den Multiplizitätsgrenzen C_1<–(1)—A—(0..*)–>C_2, so könnten beispielsweise die drei Paare $\{(1,0),(1,1),(1,n)\}$ die Menge der abzudeckenden Kombinationen repräsentieren. Konkret müssten somit folgende Situationen in den Testeingaben vorkommen: (i) C_1-Instanz ohne Verbindung zu einer C_2-Instanz, (ii) C_1-Instanz mit genau einem Link zu einer C_2-Instanz sowie (iii) C_1-Instanz mit Links zu n (mit $n > 1$) verschiedenen C_2-Instanzen.

2. ***GN*** (Generalization Criterion) – Dieses Kriterium gilt genau dann als erfüllt, wenn für jede Klasse des Klassendiagramms, welche Unterklassen besitzt, die Testmenge solche Objektdiagramme umfasst, die in ihrer Gesamtheit sicherstellen, dass für jede (konkrete) Unterklasse mindestens eine Instanz vorkommt.

3. ***CA*** (Class Attribute Criterion) – Für jede Klasse mit Attributen muss durch die Testmenge sichergestellt werden, dass pro solcher Klasse eine vorgegebene Menge an Kombinationen von repräsentativen Attribut-Wertebelegungen – berechnet durch Bildung des Kartesischen Produktes über den individuellen Mengen an repräsentativen Werten, jeweils für die einzelnen Attribute – über der Menge aller Instanzen in den Testmodellen auftritt.

   Die Mengen an relevanten Werten für ein einzelnes Attribut bestimmen sich entweder aus der konkreten Problemstellung heraus, anhand einer generischen Wertemengenpartitionierung in Abhängigkeit des Attributtyps oder durch eine Kombination der beiden Ansätze.

Die Kriterien *AEM* und *CA* sind folglich Ausprägungen des in Abschnitt 5.3.4 vorgestellten partitionsbasierten Testens, vgl. auch [OB88] oder [AO08, Kap. 4].

Ein Ansatz zum Ableiten von Testeingaben für das Testen von *UML*-basierten Software-Designs wird von Trong et al. in [DGF06] präsentiert. Gesteuert wird der Ableitungsprozess anhand der oben beschriebenen Überdeckungskonzepte aus [And+03]. Die eigentliche Testausführung kann mit Hilfe des *UMLAnT*-Werkzeugs vereinfacht werden, wie in [Tro+05] beschrieben.

Sadilek und Weißleder stellen in [SW08] ein Konzept und ein passendes Werkzeug zum Testen von *Metamodellen* nach *EMF*-Lesart vor. Ziel des Testvorhabens ist es, zu zeigen, dass das sich im Entstehen befindende, zu überprüfendes Metamodell eine passende (abstrakte) Syntax für eine Menge von exemplarischen (Test-)Modellen beschreibt. Vereinfacht gesagt beruht der Ansatz darauf, dass eine Menge gewünschter Instanzen entwickelt und generisch (ohne direkten Bezug zum zu testenden Metamodell) beschrieben wird, so dass anschließend durch Instanziierung überprüft werden kann, ob das untersuchte Metamodell tatsächlich eine Sprache beschreibt, in der die gewünschten Instanzen als valide Elemente vorkommen. Mit Hilfe eines vorgestellten Metamodells zur generischen Beschreibung potentieller Metamodellinstanzen, *Test-Metamodell* (TMM) genannt, werden die potentiellen Instanzen als Testdaten spezifiziert. Das TMM ähnelt grundsätzlich einem Metamodell zur Beschreibung von Objektdiagrammen. Ein auf JUnit basierendes Testrahmenwerk (*MMUnit*) erzeugt aus dieser Darstellung ausführbare Testfälle, in denen die einzelnen Instanziierungsabläufe überprüft werden. Bei *positiven* Testfällen sollte dies gelingen und ggf. vorhandene OCL-Bedingungen sollten erfüllt sein, bei *negativen* Tests dagegen nicht. Als Orakel dient, neben der Überprüfung der optionalen

OCL-Bedingungen, einerseits, die auf einem Namensvergleich basierende Überprüfung auf Existenz der benötigten Typen und andererseits, der Instanziierungsvorgang selbst. Letzterer könnte beispielsweise aufgrund fehlschlagender Attributzuweisungen scheitern.

Einen völlig anderen Ansatz zum Testen verfolgen Aichernig und Pari Salas, indem sie in [AS05] Mutationsüberdeckung mit *OCL*-basierten Spezifikationen kombinieren. In der Arbeit betrachten die Autoren zwar Vor- und Nachbedingungen von Operationen, denkbar wäre es eventuell aber auch, den Ansatz so abzuändern, dass Modelle zu suchen sind, welche die mutierten statischen Bedingungen entdecken können, indem ein solches Modell die eine Menge der statischen Bedingungen erfüllt und gegen die andere Menge verstößt. In einer Arbeit von Maoz et al. wird genau ein solcher Ansatz beschrieben, vgl. [MRR11c].

6.2.2 Testen von Aktivitätsdiagrammen

Wie bereits in Kapitel 2 erläutert, dienen Aktivitätsdiagramme der *UML* grundsätzlich der Beschreibung von Verhalten, insbesondere von Prozessen und Abläufen, s. auch Abbildung 2.1. Entsprechende Beschreibungen sind somit für das Testen besonders relevant, da sie zum einen als Testmodell zum anderen als Testgegenstand dienen können.

In Abschnitt 4.3 wurde dargelegt, dass der Kontrollflussanteil der *SDM*-Sprache, was die Darstellung und einige Konzepte betrifft, an Aktivitätsdiagramme angelehnt wurde. Somit erscheint es sinnvoll, sich in diesem Zusammenhang auch mit der Literatur auseinanderzusetzen, die das Testen von Aktivitätsdiagrammen zum Thema hat, trotz der Tatsache, dass einige markante Eigenschaften von Aktivitätsdiagrammen, wie (i) Parallelität, ausgedrückt durch Fork- und Join-Knoten, (ii) komplexe Verzweigungen mit *Decision*- und *Merge*-Knoten, (iii) Objektflüsse, (iv) Petri-Netz-Semantik, zur Beschreibung von Token-Flüssen, (v) Signale, (vi) spezielle Objektknoten wie «`Datastore`»-Instanzen sowie (vii) Bereiche bzw. Partitionen[6], für den in dieser Arbeit im Mittelpunkt stehenden Betrachtungsgegenstand, die *SDM*-basierten Transformationen, irrelevant sind.

Grundsätzlich legen viele Arbeiten nahe, dass Aktivitätsdiagramme in erster Linie zum modellbasierten Testen geeignet sind, vgl. z. B. [Wan+04; BLL04; Kim+07; Che+09; FFP10; Hol+11; NS11]. So lässt sich mittels entsprechender Diagramme z. B. Java- [Che+09; NS11] oder C-Code [Hol+11] testen. Allerdings ist es auch möglich, auf diese Weise *GUIs* zu testen [FFP10]. Die angeführten Ansätze unterscheiden sich zum Teil recht deutlich, z. B. im Hinblick auf die unterstützten Sprachfeatures oder die *UML*-Version.[7]

Darüber hinaus existieren aber auch Arbeiten zum direkten Testen des durch die Diagramme beschriebenen Verhaltens selbst, z. B. [CMK08; CMK10; Mij+13]. Insbesondere der in [Mij+13] erwähnte Interpreter (auf Basis einer virtuellen Maschine für *fUML* aus [13a]) stellt eine Schlüsselkomponente für das direkte Testen dar. Erst hierdurch sind Tests im üblichen Sinne der Beobachtung des Laufzeitverhaltens überhaupt möglich. Die eingesetzten Techniken – insbesondere Überdeckungskonzepte – sind allerdings nicht auf eine bestimmte Verwendungsart der Diagramme eingeschränkt, so dass sich in dieser Hinsicht die Grenzen zwischen beiden Kategorien auflösen.

[6] Auch bekannt unter der engl. Bezeichnung *Swim Lanes*.

[7] Mit dem Versionssprung der *UML* von 1.x auf 2.x haben sich die Semantik und teilweise auch die Nomenklatur der Aktivitätsdiagramme grundlegend verändert. Dabei wurde der ursprüngliche Bezug zu Zustandsautomaten durch eine Petri-Netz-artige Beschreibung ersetzt.

Wenn man im Kontext der Aktivitätsdiagramme von Überdeckung spricht, geht es typischerweise – Ausnahmen, wie der Äquivalenzklassenansatz aus [Che+05b], mal ausgeschlossen – um die systematische Traversierung und strukturelle Abdeckung des gerichteten Graphen, den das Aktivitätsdiagramm aufspannt. Es lassen sich dann die graphbasierten Überdeckungskonzepte aus Abschnitt 5.3.2 anwenden. Somit besteht dann das konkrete Ziel in der Überdeckung einzelner (i) Knoten (Activity bzw. Aktionen), (ii) Kanten (Transition bzw. Flow-Kanten) oder (iii) Pfade. Letztere können gegebenenfalls auf bestimmte Unterklassen eingeschränkt sein, z. B. wie beispielsweise in [AO08, S. 35 ff] zusammenfassend dargestellt (i) *einfache* Pfade ohne innere Schleifen, (ii) *primäre* Pfade (entsprechen einfachen Pfaden mit maximaler Länge) oder (iii) semantisch sinnvolle vorgegebene Pfade, die ein bestimmtes Benutzerverhalten beschreiben, sog. *Szenarien*.

Das Testen mittels Abdeckung *aller* Pfaden stellt auch hier eine praktisch oder sogar theoretisch unlösbare Aufgabe dar, da aufgrund von Schleifen sehr viele oder sogar unbegrenzt viele Pfade existieren können. Somit basieren alle praktikablen Verfahren auf einer *beschränkten* Menge von Pfaden. Als problematisch können sich ggf. auch die speziellen Konstrukte von Aktivitätsdiagrammen erweisen, mit denen es möglich ist, Nebenläufigkeit mittels Fork- und Join-Konzepten zu modellieren. Der dadurch resultierende Nichtdeterminismus, bezogen auf die Reihenfolge von Ereignissen (i. S. v. *Interleaving*) bei dem von außen beobachtbare E/A-Verhalten, führt ggf. zu einer großen Anzahl von möglichen Konstellationen, die beim Testen berücksichtigt werden müssten. Oft sind einige Ereignisreihenfolgen auch als äquivalent anzusehen, was beispielsweise ein Orakel entsprechend berücksichtigen müsste.

Ansätze zur Generierung von repräsentativen Szenarien zum Testen nutzen häufig Zwischendarstellungen für die zu testenden Aktivitätsdiagramme, wie beispielsweise (i) normalisierte Aktivitätsdiagramme (reduziert auf bestimmte Konstrukte) [Kim+07], (ii) spezielle gerichtete Graphen [KS09; NS11], (iii) Petri-Netze [CMK10] oder (iv) Testbäume[8] [BLL04; LL05; XLL05]. Technisch basieren Testgenerierungs- und Optimierungsverfahren i. d. R. auf:

1. Genetischen Algorithmen [JSM14],
2. Optimierungsheuristiken, wie
   a) Ameisenalgorithmus [LL05],
   b) Adaptive Agenten [XLL05],
3. Suchalgorithmen [KS09],
4. Model-Checking [CMK08; CMK10],
5. Zufallsbasierter Testgenerierung [CQL06; Che+09].

Einen interessanten Ansatz mit Bezug zum mutationsbasierten Testen stellt die Arbeit [MRR11b] dar. Maoz et al. stellen darin ein speziell an Aktivitätsdiagramme angepasstes Diffing-Verfahren namens *addiff* vor, dass auf *semantische* Unterschiede zwischen Diagrammen abzielt. Dabei bestimmt ein Werkzeug mittels Model-Checking sog. *Diff-Witnesses* in Form von möglichst minimalen Ablaufsequenzen, die beide Diagramme durch beobachtbar unterschiedliches Verhalten sicher voneinander separiert. Würde man aus einem zu testenden Aktivitätsdiagramm einen nichtäquivalenten Mutanten ableiten, könnte man ggf. mit dem Verfahren einen Testfall suchen, der den Mutanten tötet.

[8] Vgl. hierzu [Cho78].

6.2.3 Formale Verifikation bei Klassendiagrammen, Metamodellen und OCL

Um Klassendiagramme mit formalen Verifikationstechniken untersuchen zu können (bzw. um ihnen vielleicht überhaupt erst eine präzise Semantik zu geben), wird eine mathematisch exakte, formale Beschreibung benötigt. In der Literatur sind diesbezüglich unterschiedliche Ansätze bekannt. Stellvertretend sei hier auf vier Optionen verwiesen, nämlich (i) die *Z-Notation* [Eva98; Eva+99], (ii) *Beschreibungslogiken* (engl. Description Logics) [BCG05], (iii) die Aussagenlogik [Soe+10], sowie (iv) die Prädikatenlogik erster Stufe [BKS02].

Eine wichtige Eigenschaft von Klassendiagrammen und den mit ihnen verwandten, älteren *ER*-Modellen, die sich sinnvoll mittels formaler Verifikationstechniken untersuchen lässt, ist die endliche Erfüll- bzw. Instanziierbarkeit (engl. Finite Satisfiability) eines Diagramms. Ein Beispiel für eine Arbeit, die dieser Frage nachgeht, ist [MB07]. Die Autoren Maraee und Balaban setzen sich darin mit der *effizienten* Lösung des Entscheidungsproblems, auf Grundlage eines auf *Linearer Programmierung* basierenden Verfahrens, in diesem Sinne auseinander. Sie untersuchen die Erfüllbarkeit von *UML*-Klassendiagrammen ohne OCL-Nebenbedingungen, dafür aber mit sog. *Generalization-Sets*[9] der *UML*, v2.x.

Darüber hinaus existieren einige werkzeuggestützte Verfahren zur Verifikation und Validierung von *UML*-Klassendiagrammen und *OCL*-Ausdrücken:

1. *Alloy* – Bei Alloy [Jac02] handelt es sich um eine Modellierungssprache auf Grundlage einer relationalen Logik sowie um ein Werkzeug (den *Alloy-Analyzer*) zur Analyse von Strukturen, die in eben jener Sprache beschrieben sind. im Hintergrund werden die Beschreibungen und die zu überprüfenden Eigenschaften in ein *SAT*-Problem übersetzt und mittels automatisierter Lösungsverfahren untersucht. Laut der Projektseite[10] weist die Alloy-Sprache Einflüsse durch andere bekannte Modellierungssprachen auf, wie beispielsweise der *Z-Notation* oder *UML*.

   Mit *UML2Alloy* existiert ein Werkzeug zur automatisierten Übersetzung von (eingeschränkten) Klassendiagrammen inklusive OCL-Nebenbedingungen nach Alloy [Ana+07]. Dabei ist die Abbildung keineswegs trivial oder eindeutig, auch da sich die Modellierungsansätze signifikant unterscheiden. Ein auf dieser Übersetzung aufbauender Ansatz zur automatisierten Rückübersetzung von durch eine beschränkte Suche gefundenen Alloy-Modellen in Objektdiagramme wurde bereits vorgestellt [SAB10].

2. *UMLtoCSP* – Mit *UMLtoCSP* existiert ein Werkzeug zur vollständig *automatischen* Überprüfung bestimmter Eigenschaften (z. B. Erfüllbarkeit, Ausschluss der Existenz einer Subsumtionsbeziehung zwischen Bedingungen oder Lebendigkeit bestimmter Klassen) bei *UML*-Klassendiagrammen mit *OCL*-Bedingungen, vgl. [CCR07]. Der Ansatz fußt auf einer Übersetzung in *CSPs*, die mittels *Constraint Logic Programming (CLP)*, also einer Form der Logischen Programmierung (mit unmittelbarem Bezug zu *Prolog*), gelöst werden – konkret implementiert mit Hilfe des ECL^{i}PS^e-Werkzeugs.[11] Ein konkretes Anwendungsbeispiel sowie einen kurzen

[9] Vgl. [11c, S. 72].

[10] `http://alloy.mit.edu/alloy/` (abgerufen am 31.7.2014)

[11] Achtung, nicht zu verwechseln mit der Eclipse-*IDE*, vgl. auch `http://eclipseclp.org/` (zuletzt abgerufen am 31.7.2014).

Vergleich, unter anderem mit einigen der hier ebenfalls aufgeführten Verifikationsansätzen, findet sich beispielsweise in [CCR08].

Darüber hinaus stellen Gonzales et al. in [Gon+12] eine Erweiterung des Ansatzes auf *EMF*-basierte Metamodelle vor. Hierdurch ist es grundsätzlich möglich, Instanzen eines beliebigen *EMF*-basierten Metamodells abzuleiten.

3. *USE*-Tool – Im Kontext des *USE*-Werkzeugs[12] existieren zwei Ansätze zur Validierung von Klassendiagrammen mit OCL-Bedingungen. Der ältere Ansatz aus [GBR05] basiert auf einer Sprache namens *ASSL* (A Snapshot Sequence Language), mit welcher der Aufbau von Modellen beschrieben werden kann. Auf der Basis entsprechender Spezifikationen werden dann Instanzen mit vorgegebenen Eigenschaften gesucht.

   Der jüngere Ansatz aus [KHG11], *OCL2Kodkod* genannt, basiert dagegen auf der Übersetzung von *OCL*-Ausdrücken und Klassendiagrammen in eine relationale Logik mit anschließender Abbildung auf klassische Aussagenlogik und einer Analyse durch das *KodKod*-Werkzeug [TJ07]. Letzteres ist ein ebenfalls *SAT*-basierter Constraint-Solver. Die Übersetzung in ein zur Lösung geeignetes *SAT*-Problem erfolgt dabei ohne den Umweg über Alloy (*KodKod* ist, zumindest zum Zeitpunkt der Erstellung dieser Zeilen, ein integraler Bestandteil des *Alloy-Analyzer*-Werkzeugs).

4. *Isabelle/HOL-OCL* – HOLOCL, vgl. [BW08], ist eine Beweisumgebung für Klassendiagramme inklusive OCL-Ausdrücken auf Basis des interaktiven Theorembeweisers *Isabelle/HOL* [NWP02; Pau94]. Die Übersetzung der Spezifikation in die Logik des Theorembeweisers erfolgt dabei, im Gegensatz zur eigentlichen Beweisführung, vollautomatisch.

   Aufbauend auf HOLOCL, wird in [Bru+11] ein Ansatz zur Testfallgenerierung vorgestellt. Eine konkrete Umsetzung war, laut [Bru+11], zum Veröffentlichungszeitpunkt innerhalb der HOL-TEST GEN-Umgebung [BW09] geplant.

Ein weiteres interessantes und angrenzendes Themenfeld mit unmittelbarer Relevanz für die in Abschnitt 4.1.1 angesprochenen Graphgrammatiken ist die Übersetzung von *OCL*-Invarianten in sog. *globale Graph-Bedingungen*. Motivation hierfür ist die Ableitung von bzw. Nach- und Vorbedingungen von Produktionen der Graphgrammatik. Diese ermöglichen es per Konstruktion sicherzustellen, dass die Invarianten immer eingehalten werden. Eine recht aktuelle Arbeit zu diesem Themenfeld ist beispielsweise [Are+14].

6.3 Qualitätssicherung bei Modelltransformationen

Betrachten wir nun Arbeiten, die sich mit der Qualitätssicherung von Modell- und Graphtransformationen auseinander setzen. Leider ist der gesamte Themenkomplex so umfangreich, dass hier nicht alle existierenden Arbeiten zu thematisieren sind. Die Betrachtung kann also nur exemplarisch erfolgen.

Der Vorstellung konkreter Techniken bzw. Verfahren zur Qualitätssicherung soll ein Blick auf die Literatur vorausgehen, die sich mit dem Qualitätsbegriff in Bezug auf *MTs*

[12] `http://sourceforge.net/projects/useocl/` (zuletzt abgerufen am 8.1.2015).

auseinandersetzt. Beispielsweise stellen Mohagheghi und Dehlen in [MD08a] ein *Rahmenwerk zur Qualitätsbewertung* für das *MDE*-Paradigma vor. Ein wesentlicher Punkt liegt dabei in der Identifikation von *Qualitätszielen*. Sie wenden das Rahmenwerk auf *MTs* an und führen aus, dass für Transformationen zwei wesentliche Aspekte betrachtet werden müssen, nämlich der Transformations*prozess* an sich sowie die Transformations*spezifikation*; also das Modell, welches die Transformation definiert bzw. beschreibt. Für beide Aspekte werden einige Qualitätsziele genannt: hohe Performanz für den Prozess sowie Konsistenz, Wiederverwendbarkeit, Einfachheit und Minimalität für die Spezifikation. Die Ziele werden durch die Angabe von Systemkomponenten und -eigenschaften ergänzt, die großen Einfluss auf die Erfüllung der Ziele haben. Auch werden Möglichkeiten zur Untersuchung dieser Systembestandteile und -eigenschaften aufgeführt, wie z. B. Inspektionen, Messungen, werkzeuggestützte Analysen.

Van Amstel et al. beschreiben in [ALB08] ein *MT*-spezifisches Qualitätsmodell als erstrebenswertes Ziel. Die Autoren führen in diesem Zusammenhang verschiedene Qualitätseigenschaften auf (unter Verweis auf eine klassisches Buch zur Softwarequalität von Boehm et al. aus dem Jahr 1978; vgl. auch [BBL76]), die sie mit Bezug auf Transformationen als relevant erachten: (i) Verständlichkeit (engl. Understandability) – wie gut ist die Funktionalität zu erfassen, (ii) Modifizierbarkeit (engl. Modifiability) – wie gut ist die Funktionalität anpassbar oder erweiterbar, (iii) Wiederverwendbarkeit (engl. Reusability) – wie gut lässt sich die Funktionalität an anderer Stelle wiederverwenden, (iv) tatsächlich erreichter Grad an Wiederverwendung (engl. Reuse), (v) Modularität (engl. Modularity) – wie feingliedrig ist die Funktionalität strukturiert, (vi) Vollständigkeit (engl. Completeness) – wie groß ist der funktionale Umfang, (vii) Konsistenz (engl. Consistency) – wie einheitlich ist die Realisierung *oder* wie groß ist die Übereinstimmung mit der Spezifikation, (viii) Prägnanz (engl. Conciseness) – wie groß ist der Anteil an Redundanz und Überflüssigem.

In [Ams10] unterscheidet van Amstel zwischen *interner* und *externer* Qualität bei Transformationen. Dabei bezieht er die interne Qualität auf Eigenschaften der transformationsbeschreibenden Artefakte, externe Qualität dagegen auf das Verhältnis zwischen Qualitätsmaßen der Ein- und der Ausgabe der Transformation, quasi dem „*Qualitätsdelta*“. Auch wird zwischen *direkter* und *indirekter Qualitätsbewertung* unterschieden. Bei ersterer werden Kenngrößen i. S. v. *Metriken* – der nächste Unterabschnitt betrachtet diese noch ausführlicher – direkt aus der Transformationsbeschreibung ermittelt. Bei der indirekten Bewertung werden wiederum Kenngrößen der Ein- und der Ausgabe miteinander verglichen, was nicht immer sinnvoll möglich ist, wie van Amstel betont.

6.3.1 Visualisierungstechniken

Visualisierungstechniken werden in der Entwicklung technischer Systeme seit jeher eingesetzt, da sie der menschlichen Auffassung besonders entgegenkommen. Das Feld ist zu umfangreich, um hier im Detail betrachtet zu werden, deshalb wird nur eine besonders relevante Arbeit exemplarisch aufgegriffen.

In [AB11] werden, ebenfalls von van Amstel et al., *Visualisierungstechniken* als geeignetes Mittel zur Bewertung und Verbesserung der *MT*-Qualität vorgeschlagen. Diese könnten dabei helfen, Transformationen besser zu verstehen. Insbesondere bei Pflege und Weiterentwicklung von *MT*-Programmen sei das zusätzliche Verständnis wichtig. Es werden drei Zusammenhänge genannt, für die eine Visualisierung gewinnbringend erscheint,

nämlich (i) Metriken, (ii) Struktur- und Abhängigkeitsgraphen sowie (iii) die Abdeckung des Metamodells (engl. Metamodel Coverage). Letztere beschreibt, welche Elemente des Ein- und des Ausgangsmetamodells bei einer Ausführung der Transformation durch die Transformationsregeln tatsächlich referenziert werden. Daneben kann ggf. auch nachvollzogen werden, welche konkreten Modellelemente referenziert werden.

6.3.2 Maße und Metriken

Softwaremaßzahlen bzw. -metriken stellen eine wichtige Grundlage dar, um (Qualitäts-) Eigenschaften einer Software greif- bzw. messbar und damit vergleichbar zu machen, vgl. z. B. [Lig02, Kap. 6]. Wie bereits in Kapitel 5 dargelegt, wird die Testüberdeckung in Form von Metriken angegeben. Der IEEE-Standard `1061-1992` [IEE98] definiert eine Softwarequalitätsmetrik folgendermaßen:

> „A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality“.

Also als eine Funktion, die aus spezifischen Eigenschaften einer Software einen numerischen Wert ableitet, der letztendlich ihre Qualität charakterisiert. In Analogie zu Maßzahlen für klassische Software (nicht notwendigerweise mit einem Bezug zum Testen) – man denke z. B. an die *zyklomatische Komplexität* von McCabe [McC76] – wurden in der Literatur auch (Qualitäts-)Maße speziell für *MTs* vorgestellt.

Van Amstel et al. stellen in [ALB08] beispielsweise 27 Metriken – unterteilt in Metriken für Umfang bzw. Größe, realisierte Funktionalität, Module und Konsistenz – für einen spezifischen *MT*-Ansatz auf Grundlage des sog. *ASF+SDF*-Termersetzungssystems vor. Darüber hinaus setzen sie die Metriken in Beziehung zu typischen Qualitätseigenschaften, wie Verständlichkeit, Modifizierbarkeit, Wiederverwendbarkeit etc., jeweils angepasst an Modelltransformationen. Eine noch umfangreichere empirische Untersuchung dieser Zusammenhänge sowie eine Analyse, ob die Metriken zuverlässige Abschätzungen der (anhand von Expertenmeinungen bestimmten) Qualität liefern, erfolgt durch dieselben Autoren in [ALB09].

In [Vig09] greift Vignaga die Darstellungsart aus [ALB08] auf und verwendet sie, um nunmehr 81 Metriken für die *ATL*-Sprache – unterteilt in sogenannte *Unit*-, *Module*-, *Library*-, *Rule*-, *Matched Rule*-, *Lazy Matched Rule*-, *Called Rule*- und *Helper*-Metriken – vorzustellen und ebenfalls eine Einschätzung der Abhängigkeiten den Qualitätseigenschaften vorzunehmen, allerdings ohne Rückgriff auf empirische Studien, was von Rahim und Whittle in [AW13] angemerkt wird. Van Amstel erweitert zusammen mit van den Brand in [VB10] wiederum die Menge der von Vignaga vorgestellten *ATL*-spezifischen Metriken.

In [AW13] werden darüber hinaus noch weitere Arbeiten mit Bezug zu *MT*-Metriken erwähnt und miteinander sowie mit anderen Ansätzen verglichen.

6.3.3 Entwurfsmuster

Entwurfsmuster bzw. Design-Patterns beschreiben häufig genutzte und erprobte Lösungsansätze für typische Probleme in der Entwicklung von Programmen. Der Begriff wurde von Gamma et al. geprägt [Gam+95] und steht in seiner ursprünglichen Bedeutung für

> „[...] descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.“ [Gam+95, S. 3]

also für Beschreibungen miteinander interagierender Klassen und Objekte, die gemeinsam Lösungen für Design-Probleme darstellen.

Häufig vorkommende Problemstellungen lassen sich auch in *MT*-Aufgaben identifizieren und gute Lösungen in Form entsprechender Design-Patterns verallgemeinern. Als früheste Quelle, welche die Idee zur Identifikation von Design-Patterns für *MTs* beschreibt, wird regelmäßig [BJP05] genannt. In besagter Arbeit werden auch zwei exemplarische Design-Patterns anhand der *ATL*-Sprache vorgestellt: „Transformation Parameters“ und „Multiple Matching Patterns“.

Syriani und Gray betrachten in [SG12] neben anderen Aspekten auch Design-Patterns für *MTs*. Die Erstellung einer Sammlung entsprechender Muster wird als eine der Herausforderungen einer umfassenden Qualitäts*bewegung* für *MTs* identifiziert. Verschiedene Aspekte einer solchen Sammlung werden beleuchtet, wie z. B. das *Suchen und Finden* geeigneter Muster, die *formale Beschreibung* der identifizierten Muster sowie die Entwicklung einer geeigneten *Klassifikation*.

Agrawal et al. stellen in [Agr+05] zwei Design-Patterns vornehmlich für die *GReAT*-Graphtransformationssprache vor („Leaf Collector“-, „Transitive Closure“-Pattern). Iacob et al. beschreiben dagegen in [ISH08] Design-Patterns für und mit *QVT*-Relations („Mapping“-, „Refinement“- – genauer: „Relation Refinement“ und „Node Refinement“ –, „Node Abstraction“-, „Duality“- und „Flattening“-Pattern)

In [Joh+09] präsentieren Johannes et al. einen Ansatz zur teilautomatisierten Ableitung verfeinerter bzw. konkreter *Domain Specific Languages (DSLs)* aus abstrakteren *DSL*-Beschreibungen; der Anwender formuliert seine Entwicklungsartefakte dabei in einer abstrakten *DSL*. Die Autoren identifizieren in diesem Zusammenhang vier wiederkehrende Teilaufgaben, die sie in Form generischer *Abbildungsmuster* verallgemeinern. Die Umwandlung einer abstrakten in eine verfeinerte *DSL* lässt sich dann als Komposition von Teilabbildungen auffassen, jeweils beschreibbar durch die zuvor entwickelten Abbildungsmuster.

Lano und Kolahdouz-Rahimi unterscheiden in [LK11b] dagegen zwischen Mustern für die Spezifikation und für die Implementierung von *MTs*. Sie identifizieren insgesamt vier generische Spezifikationsmuster („Conjunctive Implicative Form“, „Recursive Form“, „Auxiliary Metamodel“, „Construction and Cleanup“) und drei Implementierungsmuster („Phased Creation“, „Unique Instantiation“, „Object Indexing“).

In [ES13] stellen Ergin und Syriani ein Design-Pattern im Kontext der *MoTif*-Sprache vor („Fixed-Point Iteration“). Aufbauend auf dieser Arbeit wird von denselben Autoren die *DelTa*-Sprache zur Beschreibung von Design-Patterns für *MTs* in [ES14b; ES14a] eingeführt. Mit Hilfe der Sprache werden vier anderweitig veröffentlichte Design-Patterns beschrieben und diese zu Demonstrationszwecken jeweils in fünf *MT*-Sprachen (*MoTif*, *AGG*, *Henshin*, *Viatra2*, *GrGen.NET*) exemplarisch umgesetzt. Die Arbeit umfasst auch Beschreibungen der jeweiligen Umsetzungen.

Einen leicht anderen Fokus als die zuvor erwähnten Arbeiten hat die Arbeit [Cua+09] von Cuadrado et al.. Zum einen wird mit *OCL* keine klassische Transformationssprache betrachtet. Zum anderen stehen hier weniger Design-Überlegungen im Vordergrund als Optimierungen zur Verbesserung der Laufzeitperformanz. Diese Optimierungen werden ebenfalls in Musterform verallgemeinert.

In der umfangreichen Klassifikation [LK14] von Lano und Kolahdouz-Rahimi werden fünf Muster-Kategorien für *MT*-Programme definiert. Die Unterscheidung erfolgt anhand des durch sie verfolgten Ziels: (i) „Rule modularization patterns“ – Verbesserung der Aufteilung von Funktionalität auf *MT*-Regeln sowie des Designs, (ii) „Optimization patterns“ – Muster zur Optimierung (z. B. der Performanz oder Effizienz), (iii) „Model-to-text patterns“ – *M2T*-spezifische Lösungsansätze, (iv) „Expressiveness patterns“ – Simulation fehlender Konstrukte durch vorhandene, (v) „Architectural patterns“ – Optimierung der Architektur (z. B. durch die Anpassung von Metamodellen) oder von Abläufen. Auch werden konkrete Muster aus verschiedenen Quellen zusammengetragen, beschrieben und entsprechend kategorisiert. Die Kombinierbarkeit von Mustern, Kriterien für deren Auswahl sowie die Erprobung und Bewertung der Muster im Rahmen einer Evaluation werden ebenfalls betrachtet.

6.3.4 Formale Verifikation von Modelltransformationen

In diesem Abschnitt wird versucht, einen Überblick über existierende Ansätze zur formalen Verifikation von *MTs* anhand einiger exemplarischer Nennungen zu geben. Obwohl die Übergänge fließend sind, werden Ansätze, die überwiegend bzw. ausschließlich der formalen Verifikation von *Graph*transformationen gewidmet sind, in einem eigenen Unterabschnitt gesammelt vorgestellt. Für einen Überblick gängiger Verifikationsansätze im Bezug auf *MTs* und *GTs* sei auch auf die Übersichtspapiere von Amrani et al. [Amr+12], Calegari und Szasz [CS13; CS12] sowie Ab Rahim und Whittle [AW13] verwiesen.

In [Amr+12] unterscheiden die Autoren beispielsweise drei Dimensionen, anhand derer sie die existierenden Arbeiten klassifizieren: (i) Transformationsgrundlagen (Definitionen und Auffassungen des Begriffs, Sprachfamilien, existierende Klassifikationen), (ii) wichtige (Qualitäts-)Eigenschaften von *MTs* und der eingesetzten Sprache (*Terminierung*, *Determiniertheit* bzw. *Eindeutigkeit der Ausgabe*, *Konfluenz*, *Typisierung*) und der Transformationsaufgabe (*Konformitätsbedingungen*, *Vollständigkeit*, *Bisimilarität* etc.), (iii) Techniken zur formalen Verifikation, wobei (iii.a) transformations- und eingabe*unabhängige* (manuelle Beweise, Abbildungen auf existierende Formalismen), (iii.b) transformationsabhängige und eingabeunabhängige (Model-Checking, statische Analysen, Theorembeweiser) und (iii.c) transformations- und eingabe*abhängige* (Traceability-Link-basierte Verfahren, Petri-Netze, CSP-Solver) Ansätze unterschieden werden.

Calegari und Szasz greifen diese Art der Darstellung auf und verfeinern sie in [CS13] um (i) eine weitere Qualitätseigenschaft für Sprachen (Erhalt der Ausführungssemantik), (ii) zwei weitere Eigenschaften bezogen auf die Transformationsaufgabe (Funktionalität[13] und syntaktische Vollständigkeit[14]) sowie (iii) die Berücksichtigung der Techniken *Testen* und *Korrektheit-durch-Konstruktion*. Ein weiterer Beitrag liegt in der Klassifizierung weiterer Arbeiten, vgl. auch den zugehörigen technischen Bericht [CS12].

Ab-Rahim und Whittle betrachten in [AW13] im Gegensatz zu Amrani et al. explizit auch nichtformale Verifikationsansätze wie Metriken und testende Verfahren. Zusätzlich nehmen sie auch weitere Aspekte, wie z. B. die verfügbare Werkzeugunterstützung, mit in ihre Betrachtungen auf. Dazu wird der Stand der Forschung in Form von insgesamt 57 unterschiedlichen Ansätzen untersucht und anhand der folgenden fünf Katego-

[13] Rechtseindeutigkeit bei gleichzeitig linkstotalem Verhalten.
[14] Definiert über eine vollständige Metamodellüberdeckung.

rien klassifiziert: (i) Testansätze, (ii) Ansätze auf der Grundlage von Theorembeweisern, (iii) Ansätze auf der Grundlage der Theorie von Graphtransformationen, (iv) Model-Checking-basierte Ansätze sowie (v) Ansätze auf der Basis von Inspektionen und/oder Metriken. Für jeden Ansatz werden Informationen zur genutzten Technik, dem Grad der Formalisierung, dem damit verbundene Aufwand, der Werkzeugunterstützung, den zu überprüfenden Eigenschaften, der unterstützten *MT*-Sprache sowie der Art der *MT* (*M2M*, *M2T*, beides) aufgeschlüsselt. Die Darstellung ist überaus umfangreich und, trotz der Breite, recht detailliert.

Lano et al. identifizieren in [LKC12] als wesentliche, nachzuweisende Eigenschaften von *MTs* deren (i) *Terminierung*, (ii) *Konfluenz* sowie (iii) *Korrektheit*. Bezüglich des letztgenannten Punktes sind, wie in [LC08] dargelegt, die *syntaktische* und die *semantische* Korrektheit zu unterscheiden. In [LKC12] werden drei grundlegende Ansätze zur Verifikation unterschieden, nämlich (i) syntaktische Analysen, (ii) die Abbildung auf einen Formalismus und (iii) die Abbildung der *MT*-Spezifikation in einem Formalismus *plus* die Ableitung einer Implementierung hieraus (als „*correct-by-construction*“ bezeichnet). Außerdem werden exemplarisch drei konkrete Techniken verglichen, nämlich die syntaktische Analyse, eine Abbildung auf die *B*-Sprache sowie eine Abbildung auf *Z3*. Die zwei letztgenannten gehören zur Gruppe (ii) und nutzen sog. *Verifikationsmodelle*.

Model-Finding-Ansätze

Viele weitere Arbeiten verfolgen ebenfalls den Ansatz, die Transformationsspezifikation oder -implementierung ganz oder teilweise in einem geeigneten Formalismus zur Analyse zu übersetzt. Beispielsweise können Modelltransformationen und nachzuweisende Eigenschaften mit Hilfe sogenannter *Transformation-Models* beschrieben werden, vgl. z. B. [Cab+10c; Büt+12a; Sel+13]. Dabei handelt es sich um eine kombinierte Darstellung der Quell-, Ziel- und Transformationssprache (inklusive eventuell vorhandener Nebenbedingungen), Vor- und Nachbedingungen der Transformationsschritte bzw. -regeln sowie Invarianten der Transformation. Ausgehend davon können dann Model-Finding-Ansätze für *UML* bzw. *MOF*-basierte Metamodelle, ggf. inklusive *OCL*-Bedingungen, genutzt werden, um Eigenschaften wie Erfüllbarkeit oder Vollständigkeit[15] nachzuweisen. Beispielsweise lassen sich das bereits zuvor erwähnte *UMLtoCSP*, wie in [Cab+10c], oder auch *UML2Alloy*, wie in [Büt+12a], nutzen, um durch das Finden von Gegenbeispielen Aussagen zu falsifizieren.

In [BEC12] beschreiben Büttner et al. einen weiteren Ansatz auf Basis von Transformationsmodellen. Darin wird die Semantik einer Teilmenge von ATL mit Hilfe eines Formalismus beschrieben, für den Korrektheitsaussagen mit Hilfe eines *SMT*-Solvers untersucht werden. Das Verfahren wird genutzt, um für eine Transformation die Eigenschaft nachzuweisen, dass für jede valide *MT*-Eingaben auch eine valide Ausgabe entsteht.

Anastasakis et al. skizzieren in [ABK07] einen Ansatz, bei dem sie relationale Modellabbildungen, genauer die dazugehörigen Regeln, in eine Spezifikation des zuvor bereits erwähnten *Alloy* übersetzen, um diese dann mit Hilfe des *Alloy*-Werkzeugs zu analysieren. Technisch basiert die Arbeit auf dem ebenfalls bereits erwähnten *UML2Alloy*, vgl. [Ana+07], desselben Hauptautors. Dazu werden einerseits die Metamodellen in eine *Alloy*-Darstellung übersetzt, andererseits werden auch die Abbildungsvorschriften der

[15] Ggf. eingeschränkt auf endliche Modellgrößen.

Transformationsbeschreibung zwischen Ein- und Ausgabeseite in eine *Alloy*-Darstellung überführt. Mit Hilfe dieses Ansatzes lassen sich verschiedene Eigenschaften bzgl. der Ausgabeseite überprüfen. Die Autoren identifizieren allerdings die geringe Skalierbarkeit als größten Hemmschuh ihres Ansatzes.

Logikbasierte Ansätze

Neben den Verifikationsansätzen mit starkem Bezug zum Model-Finding, existieren weitere Verfahren, die auf einer Kodierung der Transformation sowie der zu verifizierenden Eigenschaften in einer geeigneten Logik aufbauen. Beispielsweise verwenden Braga et al. in [Bra+11] sog. *Beschreibungslogik* (engl. *Description Logic*) um *Konsistenz*, definiert als Erfüllbarkeit des mathematischen Modells in der gewählten Logik, für die betrachteten *MTs* nachzuweisen, welche in Form von Transformationsverträgen spezifiziert sind. Folglich werden hierbei für *MTs* ebenfalls *Transformation-Models* abgeleitet und durch formales Schließen analysiert.

Boronat et al. benutzen in [BHM09] sog. *Rewriting Logic* als Zielformalismus für *MOF*, *OCL* und *QVT*. Sie sind so in der Lage, Eigenschaften von *QVT*-Transformationen mit Hilfe des *Maude-Systems*[16] zu verifizieren. In [TV10; TV11] entwickeln Troya und Vallecillo eine formale Beschreibung der *Atlas Transformation Language (ATL)*-Semantik ebenfalls für *Maude*.

Verifikation mit Petri-Netzen

Einen weiterer Zielformalismus, der zur Verifikation von *MTs* benutzt werden kann, bilden *Petri-Netze*. So können beispielsweise Petri-Netze mit Farben [Jen81], engl. *Colored Petri Nets (CPNs)*, einerseits zum Nachweis wichtiger Transformationseigenschaften (Lebendigkeit, Konfluenz), andererseits zur Fehlerlokalisierung, wie von Wimmer et al. in [Wim+09] beschrieben, verwendet werden.

De Lara und Guerra stellen in [LG09] eine Formalisierung der Semantik von in QVT-Relations geschriebenen *MTs* vor, die ebenfalls auf *CPNs* basiert. Sie zeigen, wie ein solches *CPN* (mit Hilfe der *CPNTools*[17]) analysiert und animiert werden kann. Dadurch ist es u. a. möglich, wichtige Eigenschaften der QVT-Transformation zu verifizieren (wie z. B. Terminiertheit, Konfluenz oder Konflikte zwischen Teilabbildungen) bzw. die Transformation während der Entwicklung zu validieren.

Graphtransformationen als Hilfsmittel zur Verifikation

Um dynamische Konsistenzeigenschaften zwischen Ein- und Ausgabe zu zeigen, nutzen Varró und Pataricza in [VP03] *Model-Checking*-Techniken auf Basis einer Beschreibung der dynamischen Semantik von Ein- und Ausgabemodell mittels Transitionssystemen. Letztere werden aus Graphtransformationsbasierten Beschreibungen mit Hilfe eines Werkzeugs automatisiert abgeleitet. Für eine Beispieltransformation, die Statecharts auf Petri-Netze abbildet, kann so z. B. eine als *Safety* bezeichnete Eigenschaft – sie umschreibt, dass in der Petri-Netz-Darstellung ausgeschlossen ist, dass zwei nicht-parallele Zustände gleichzeitig aktiv sind – nachgewiesen werden. Varró optimiert und erweitert

[16] `http://maude.cs.illinois.edu/` (zuletzt abgerufen am 30.9.2014)
[17] `http://cpntools.org/` (zuletzt abgerufen am 12.1.2015)

diesen Ansatz wiederum in [Var04] nochmals. Er wendet ihn auch direkt auf Modellebene an, also auf dem Transitionssystem, das sich durch problemspezifische *GT*-Regeln ergibt.

Ein häufig benötigter, wichtiger Teilschritt bei einer möglichst automatisch ablaufenden *MT*-Verifikation liegt, wie weiter oben ggf. bereits erkennbar, in der Abbildung der *MT*-Spezifikation in ein formales Analysemodell (in der Sprache des Zielformalismus). Diese Abbildung selbst kann wiederum als eine Transformation aufgefasst werden, die frei von Fehlern sein sollte, um die Gültigkeit der gewonnenen Erkenntnisse nicht zu gefährden. Um die Korrektheit einer solchen Abbildung nachzuweisen, nutzen Narayanan und Karsai in [NK08; KN08] die Möglichkeiten der *GT*-Sprache *GReAT* aus, die sogenannte *Cross-Links* zwischen den Elementen der Ein- und der Ausgabesprache im Rahmen der Ausführung der *MT* anlegen kann. Mit Hilfe dieser Links lässt sich dann die *Äquivalenz* zwischen Spezifikation und Analysemodell, z. B. im Sinne von *Bisimilarität*, nachweisen.

In [Gie+06] nutzen Giese et al. Beschreibungen auf der Grundlage von *Triple Graph Grammars (TGGs)* [Sch95], um *M2T*-Transformationen zu verifizieren. Dabei ist die semantische Äquivalenz zwischen Eingabe (Modell) und Ausgabe (Text) zu zeigen. Dies wird erreicht, indem aus der *TGG*-Beschreibung sowie der Einzelsemantiken von Quell- und Zielseite eine Beschreibung für den Theorembeweiser *Isabelle/HOL* abgeleitet wird, die anschließend teilautomatisiert bewiesen werden kann.

Bares et al. argumentieren in [BEH07], dass ein Rückgriff auf *GTs*, sowohl zur Beschreibung der dynamischen Semantik der Ein- und Ausgabemodelle als auch zur Beschreibung der *MT* zwischen den Ein- und Ausgabesprachen selbst, für die Verifikation von *MT* günstig ist. Hierdurch ist es möglich, die nachweisbaren Eigenschaften von *GTs* auszunutzen, um so den Erhalt semantischer Eigenschaften der beteiligten Modelle durch die Transformation zu garantieren. Konkret wird dies anhand der Umwandlung von zentralen in verteilte *BPEL*-Prozesse gezeigt.

In [EE08] beschreiben Ehrig und Ermel einen Ansatz zur Verifikation der *semantischen Korrektheit* sowie der *semantischen Vollständigkeit* von *MTs* zwischen Sprachen mit jeweils einer formalen Semantik. Die operationale Semantik der beteiligten Sprachen wird dazu jeweils in Form von *GT*-Regeln, den sog. *Simulation Graph Rules* (SGRs), beschrieben. Das Resultat einer *MT*, der sog. *Rule Transformation* (RT), welche die SGRs der Eingabesprache in Regeln für die Ausgabesprache unter Beachtung der zu verifizierenden *MT* zwischen den Sprachen übersetzt, wird dazu mit den eigentlichen SGRs der Ausgabesprache verglichen. Für die RT bzw. die *MT* werden entsprechende Bedingungen formuliert, die wiederum Voraussetzung für Korrektheit und Vollständigkeit sind.

Proofs-as-Model-Transformations

Eine interessante, wenn auch, wie in [AW13] festgestellt, nur eingeschränkt praxistaugliche Möglichkeit, um zu bewiesenermaßen korrekten Transformationen per Konstruktion zu kommen, ist der sog. *Proofs-as-Model-Transformations*-Ansatz von Poernomo [Poe08]. Hierin werden die Spezifikation einer *MT*, bestehend aus einer Menge von Vor- und Nachbedingungen, Invarianten sowie den beteiligten Metamodellen in eine Darstellung überführt, die geeignet für eine maschinengestützte Beweisführung ist. Anhand dieser Darstellung und einem erfolgreich durchgeführten (konstruktiven) Existenzbeweis, kann direkt aus dessen Einzelschritten die konkrete Implementierung der Transformation synthetisiert werden. Für die konkreten Details sei auf diese Arbeit verwiesen.

Verifikation von Graphtransformationen

Bei Graphtransformationen spielt, wie bereits in Abschnitt 4.2.1 dargelegt, die Formalisierung eine zentrale Rolle. Deshalb sind *GT* selbst als geeigneter Formalismus anzusehen, auf dessen Grundlage Transformationseigenschaften mathematisch exakt nachgewiesen werden können. Vielfach wurden und werden theoretische Eigenschaften der *GT*-nutzenden Sprachen bzw. der ihnen zugrunde liegenden Formalismen (in Verbindung mit dem genutzten Graphenmodell) untersucht. Zwei Fragestellungen sind dabei von besonderem Interesse, vgl. hierzu z. B. [Küs06]:

1. Terminiert das Graphtransformationssystem? Im Allgemeinen ist diese Entscheidung zwar nicht möglich, wie von Plump in [Plu98] gezeigt. Es existieren allerdings Ansätze, die diese Frage näherungsweise oder für eingeschränkte Graphtransformationssysteme beantworten können, beispielsweise auf Grundlage von (i) monotonen Funktionen (in Anlehnung an [DM79]), vgl. z. B. [Bot+05], (ii) *Layered Graph Grammars* (ursprünglich eingeführt in [RS97]), vgl. hierzu z. B. [Ehr+05], oder (iii) Petri-Netzen [Var+06].
2. Falls das System terminieren sollte, ist dann das Ergebnis für alle Eingaben *eindeutig*, das System also – unabhängig von der Reihenfolge von Regelanwendungen – *konfluent*? Hierbei ist grundsätzlich zu untersuchen, ob die Anwendungsreihenfolge der Regeln einen Einfluss auf das Ergebnis haben kann. Aufgrund des *Lemmas der Kritischen Paare*, vgl. hierzu [Plu93; HKT02; Ehr+04], lässt sich die Frage nach Konfluenz durch die Untersuchung einer endlichen Menge sog. *Kritischer Paare*, engl. *Critical Pair Analysis*, beantworten.

Daneben sind Nachweise problemspezifischer Eigenschaften auch bei *GT* von Interesse. Beispielsweise könnte zu zeigen sein, dass durch die Anwendung von einzelnen Regeln oder Regelsequenzen eine bestimmte Bedingung im Modell nie verletzt wird. Oder für einen komplexen Ablauf eines Transitionssystems, bei dem Übergangsschritte durch Regeln beschrieben werden, ist nachzuweisen, dass dieser nicht vorzeitig abbrechen kann bzw. eine Verklemmung des Systems ausgeschlossen ist.

In [BCK01] beschreiben Baldan et al. eine statische Analysetechnik für (Hyper-)Graph-Ersetzungssysteme auf Basis einer als *Petri Graphs* bezeichneten Formalisierung. Dazu wird aus einem gegebenen Startgraphen im Zusammenspiel mit den Ersetzungsregeln systematisch die *endliche* Petri-Graph-Struktur konstruiert. Letztere besteht aus einem (Hyper-)Graphen, der die erreichbaren Systemkonfigurationen beschreibt, und einem Petri-Netz, das über den Graphelementen aufgespannt ist. Bestimmte Eigenschaften lassen sich dann anhand der Analyse des Petri-Netzes bzw. des Petri-Graphen zeigen. Hierzu wurde außerdem das AUGUR-Werkzeug[18] [KK05] entwickelt und vorgestellt, mit dem sich entsprechende Analysen automatisieren lassen.

Cabot et al. übersetzen Graphtransformationsregeln dagegen in *OCL*-Bedingungen (auf Basis entsprechender Klassendiagramm-Darstellungen) [Cab+08; Cab+10a], um diese dann mit Hilfe von Werkzeugen zur Analyse von *OCL* zu verifizieren. Im konkreten Fall wird *UMLtoCSP* genutzt. Damit sind die Autoren in der Lage, verschiedenste Eigenschaften der *GT*-Beschreibung nachzuweisen, wie z. B. die Anwendbarkeit von Regeln, die Konfliktfreiheit oder Abhängigkeit zwischen Regelpaaren und die Ausführbarkeit im Sinne einer Kompatibilität zwischen Regelnachbedingungen und Modellinvarianten. Dabei werden sowohl der *SPO*- als auch der *DPO*-Ansatz unterstützt. Als Primärquelle für

[18] Für eine Beschreibung der aktualisierten Version, Augur 2, s. [KK08].

die zugrunde gelegte Idee, Graphtransformationsregeln in *OCL* zu überführen, nennen Cabot et al. eine Arbeit von Büttner und Gogolla [BG06].

Der bereits zuvor erwähnte und auf Model-Checking basierende Ansatz von D. Varró [Var04] kann auch genutzt werden, um konkrete Transformationsbeschreibungen, die in *GT*-Form gegeben sind, direkt in ein Model-Checking-Problem zu übersetzen, statt die Definition der dynamischen Semantik der Sprache auf „Metaebene“ zu übersetzen. Durch diese Art der Abbildung ist der Nachweis semantischer Eigenschaften von *GTs* (z. B. Sicherheit, Verklemmungsfreiheit etc.) mit Hilfe des *CheckVML*-Werkzeugs [SV03] möglich.[19]

Ein weiterer bekannter Model-Checking-Ansatz für *GTs* stammt von Rensink [Ren03]. Dazu werden die Übergangsfunktionen eines Graphtransformationssystems mittels *GT*-Regeln der *SPO*-Interpretation beschrieben. Mit Hilfe einer sog. *Linear-Time Temporal Logic (LTL)* (zweiter Stufe, vgl. [Ren03]) können dann zu verifizierende Eigenschaften des Systems formuliert und werkzeuggestützt analysiert werden. Hierzu wurde das *GROOVE*-Werkzeug (**GR**aphbased **O**bject-**O**riented **VE**rification) von Rensik et al. entwickelt, vgl. [Ren04], das ohne Übersetzung in ein externes Model-Checking-Werkzeug auskommt. Die beiden Model-Checking-Ansätze, *GROOVE* und *CheckVML*, wurden bereits in einer gemeinsamen Arbeit von Rensik, Schmidt und Varró miteinander verglichen [RSV04].

In [Hec98] beschreibt Heckel, wie sich für Teilspezifikation von *GTs*, den sog. *Views*, isoliert nachgewiesene, temporale Eigenschaften auf die per Komposition der Teilspezifikationen entstandene Gesamtspezifikation übertragen lassen. Damit legt er u. a. auch die theoretische Basis für die zuvor erwähnten praktischen Model-Checking-Ansätze.

In ihrer Arbeit [BS06] finden Baresi und Spoletini in *Alloy* ebenfalls einen geeigneten Zielformalismus zur Verifikation von *GTs*. Sie beschreiben exemplarisch ein verallgemeinerbares Encoding für AGG-basierte Graphtransformationen. Neben transformationsspezifischen Eigenschaften lassen sich generische Eigenschaften überprüfen. Entsprechende Eigenschaften sind (i) der Nachweis der Erreichbarkeit einer Konfiguration (bei Vorgabe der maximalen Anzahl an Ableitungsschritten) und (ii) der Nachweis der Durchführbarkeit einer Sequenz von Regelanwendungen. Außerdem besteht die Möglichkeit zur Berechnung der möglichen Konfiguration nach n Regelanwendungen (eingeschränkt durch die stark anwachsende Zahl an Optionen und notwendigen Berechnungen). Im Gegensatz zu dem Encoding aus der erwähnten Arbeit [ABK07] von Anastasakis et al., berücksichtigen Baresi und Spoletini explizit Ableitungs*sequenzen* (als Sequenzen von Graphen).

Model-Checking und Alloy-basierte Ansätze haben den Nachteil, dass sie nur Aussagen für Modelle einer limitierten Größe treffen können. Sind Beweise abschließend und für größenunbeschränkte Modelle notwendig, eignen sie sich nur dann, wenn ein Beweis durch ein Gegenbeispiel endlicher (und, aus praktischen Gründen, geringer) Größe erbracht werden kann, oder unendliche Strukturen durch Abstraktion auf endliche Gebilde abgebildet werden können. Strecker stellt dagegen in [Str08] einen ersten Ansatz vor, bei dem *GTs* in Eingaben für den Theorembeweise *Isabelle* [Pau94] umgewandelt werden. Damit wird obiges Problem umgangen, so dass ggf. mehr und andere strukturelle Eigenschaften der Graphen im Laufe einer Transformation nachzuweisen sind. Attribute werden dabei allerdings explizit ausgeklammert.

[19] Technisch basiert der *CheckVML*-Ansatz auf einer Übersetzung in eine Promela-Spezifikation für den Model-Checker *SPIN*. (Für Informationen zu SPIN s. [Hol97].)

Da Costa und Ribeiro schlagen in [CR09; CR12] vor, Graphgrammatiken ebenfalls in einen Formalismus zu übersetzen, der abschließende Beweise ermöglicht. Die Grundlage bilden Beschreibungen mit Relationen (zur Beschreibung der eigentlichen Grammatikstruktur) und Logik (zur Beschreibung von Regelanwendungen). Die Autoren schlagen vor, ihren Ansatz zur Analyse verteilter Systeme zu nutzen. Konkrete Eigenschaften – z. B. im Hinblick auf die erreichbaren Zustände – lassen sich mit Hilfe von Induktionsbeweisen verifizieren, wobei ein Einsatz von Theorembeweisern angedeutet wird. Auf Werkzeugseite wird auch hier *Isabelle* [Pau94] erwähnt.

In [PP12; PP10] beschreiben Poskitt und Plump ein Beweissystem zum Nachweis der Korrektheit von Programmbeschreibungen in der *GP*-Sprache[20] [Plu09]. Sie entwickeln dazu einen dem Hoare-Kalkül [Hoa69] vergleichbaren Kalkül. Die exemplarische Verwendung, also wie man konkrete Eigenschaften beweisen kann, ist ebenfalls Teil der Präsentation.

Die bis jetzt erwähnten Arbeiten gehen stets von einer nachträglichen Verifikation, im Sinne eines Nachweises von Eigenschaften des resultierenden Systems bzw. der Graphtransformationsbeschreibung nach einer Realisierung, aus. Eine völlig andere Herangehensweise verfolgen proaktive Ansätze, welche die einzuhaltenden Bedingungen per Konstruktion zu garantieren versuchen. Heckel und Wagner beschreiben in [HW95] beispielsweise eine Konstruktion, mit deren Hilfe globale, durch Graphmuster beschreibbare Konsistenzbedingungen, also gewisse Invarianten, schrittweise erst in Nachbedingungen für einzelne Regeln und dann in Vorbedingungen der Regeln, auch als *Anwendungsbedingungen* bezeichnet, umgewandelt werden können. Hierdurch kann garantiert werden, dass die modifizierte Regelmenge nie zu einer Verletzung der ggf. sicherheitsrelevanten Konsistenzbedinungen führen kann. Der Ansatz berücksichtigt allerdings nur graphartige Konsistenzbedingungen, keine Attribute, und nur Transformationen, die sich als Auswertungen einzelner isolierter Regeln oder schrittweiser Sequenzen dieser beschreiben lassen. Eine praktische Umsetzung einer solchen Konstruktion wird von Deckwerth und Varró in [DV14b] beschrieben.

In [DV14a] stellen Deckwerth und Varró eine erweiterte Konstruktion von Vorbedingungen aus Invarianten vor. Diese berücksichtigt Attribute sowie deklarative und von mehreren Attributwerten abhängige Attributbedingungen. Es wird darüber hinaus eine effiziente Umsetzung durch eine flexible Pattern-Matcher-Komponente beschrieben.

Einen grundsätzlich vergleichbaren Ansatz beschreiben auch Cabot et al. in [Cab+10b]. Hier werden allerdings aus Nachbedingungen *OCL*-basierte Vorbedingungen als Anwendungsbedingungen für *GTs* abgeleitet. Der Vorteil dieses Ansatzes liegt in der vereinheitlichten Behandlung von Metamodell-Nebenbedingungen (in *OCL* formuliert) und Graphbedingungen für Invarianten. Ein Nachteil ist die explizite Festlegung auf eine feste Modelltraversierung aufgrund des *OCL*-Ausdrucks. Dies kann im Hinblick auf die Ausführungsperformanz problematisch sein, wie von den Autoren in [DV14a] bemerkt.

Für weitere Arbeiten, die sich beispielsweise mit dem Nachweise der *Korrektheit* und *Vollständigkeit* bei *TGGs* auseinander setzten, sei hier auf die einschlägige Literatur verwiesen.

[20] *GP* steht für *Graph Programs*, s. [Plu09].

6.3.5 Testen von Modelltransformationen

Nach der Betrachtung *formaler* Verifikationstechniken für Modell- und Graphtransformationen werden nun testende Verfahren vorgestellt. Testverfahren werden hier in Anlehnung an die weit verbreitete Auffassung, vgl. z. B. [AW13], vornehmlich als eine Form der Verifikation aufgefasst. Testende Verfahren und Verfahren der formalen Verifikation sind dabei oft als orthogonal zueinander zu sehen. Beide Ansätze lassen sich gut und sinnvoll miteinander kombinieren. Beispielsweise können formale Verifikationsansätze im Zuge der Entwurfsphase genutzt werden, testende Verfahren dagegen im Rahmen der Implementierungsphase, wie beispielsweise von Darabos et al. angedeutet [DPV08]. Wie bereits erwähnt, argumentieren Fleurey et al. in [FSB04], dass traditionelle Testansätze für allgemeine Programme nicht ausreichen, da sie weder das potentiell erhöhte Abstraktionsniveau von *MTs* berücksichtigen noch deren Bezug zu Metamodellen. Somit werden angepasste Verfahren für *MT*-Programme benötigt.

Typische Herausforderungen des Testens von *MTs* untersuchen beispielsweise Baudry et al. in [Bau+06]. Sie identifizieren zwei Hauptaufgaben, die ein Testansatz beachten sollte: (i) die interaktive und/oder automatisierte *Generierung von Testdaten* – insbesondere auf Basis der Überdeckung eines Metamodells (Partitionsüberdeckung, Objekt- oder Modellfragmentüberdeckung) bzw. von repräsentativen Werten, welche anhand von *OCL*-Bedingungen oder der Implementierungssprache (mittels Type-Checking-Komponenten) abgeleitet werden, vgl. auch [FSB04] – sowie (ii) das Finden bzw. Erstellen eines geeignet erscheinenden *Orakels*. Letzteres z. B. anhand von Verträgen und Zusicherungen bzw. deren konkreten Repräsentationen in *OCL*.

Baudry et al. greifen in einer zweiten Arbeit [Bau+10] abermals die Herausforderungen bei einem Testansatz auf. Sie führen, neben den beiden zuvor schon genannten Aufgaben der Testdaten- bzw. Modellerzeugung sowie der Orakelerstellung, noch die Wahl eines geeigneten Adäquatheitskriteriums an. Dies setzt voraus, dass auch eine entsprechende Messbarkeit gegeben ist. Darüber hinaus bestimmen die Autoren typische Eigenschaften von *MT*-Programmen, die das Testen und insbesondere die Durchführung der drei genannten Aufgaben negativ beeinflussen können. Im Detail sind dies (i) die Komplexität der Ein- und Ausgabedaten bzw. Ein- und Ausgabemodelle, welche die Erzeugung zueinander konsistenter Modellpaare in ausreichender Anzahl erschwert, (ii) eine suboptimale Werkzeugunterstützung, welche die manuelle Ableitung von Testeingaben und die Bewertung von Testausgaben erschweren kann, sowie (iii) die grundsätzliche Vielfalt an *MT*-Sprachen und -Ansätzen, welche die Übertragbarkeit von Testansätzen einschränken kann.

Selim et al. greifen die von Baudry identifizierten drei Hauptaktivitäten beim Testen in [SCD12] auf. Sie ergänzen die Testausführung inkl. der Ergebnisbewertung als weitere Phase. Anhand der dann insgesamt vier Phasen geben die Autoren eine Übersicht anderer Arbeiten zum Testen von *MT*-Programmen.

In der bereits erwähnten Übersichtsarbeit [AW13] von Ab Rahim und Whittle ist ebenfalls ein eigener Abschnitt einer Übersicht anderer Arbeiten aus dem Testkontext gewidmet. Es werden Arbeiten zu den Themenfeldern (i) Testgenerierung, (ii) Orakelentwicklung sowie (iii) Testrahmenwerke genannt und Unterschiede herausgearbeitet. Neben einer Vorstellung und Beschreibung werden die Ansätze auch mit anderen (formalen) Verifikationsansätzen (für *MTs*) verglichen.

Berührungspunkte zu verwandten Testansätzen

Insgesamt lassen sich fünf Hauptströmungen identifizieren, welche vor allem frühe Arbeiten zum Testen von *MTs* beeinflusst haben. Im weiteren Verlauf des Kapitels wird auf diese an geeigneten Stellen verwiesen. Im Detail handelt es sich um:

1. Testansätze für *UML*-Modelle, insbesondere solche für Klassendiagramme, wie diejenigen aus [And+03; MP05], vgl. auch Abschnitt 6.2.1, finden ihre Entsprechung im Konzept der Metamodell-Überdeckung beim Testen von *MTs* (z. B. auf Seite der Eingabesprache), vgl. hierzu auch [FSB04].

2. Ansätze zum spezifikationsbasierten Testen [ROT89] sowie für *Design-By-Contract* [Mey92] – beispielsweise in der Ausprägung *JML* [LBR99] – aber auch entsprechende Werkzeuge (z. B. *Korat* [BKM02] oder *AutoTest* [CL05]) hatten und haben Einfluss auf die Entwicklung *OCL*-basierter Testansätze, vgl. hierzu beispielsweise die Arbeiten von Cariou et al. [Car+04a; Car+04b; Car+09] oder ebenfalls [FSB04].

3. Techniken aus dem Bereich des Compiler-Testing, vgl. hierzu z. B. [KP05], weisen eine gewisse inhaltliche Nähe inkl. teilweiser Überlappung bezogen auf Testverfahren für *M2T*-Transformationen und Testansätze für Codegeneratoren auf, vgl. z. B. [SC03; BKS04].

4. Testansätze, die sich auf die Qualitätssicherung von *MT*-Werkzeugen, insb. auch Transformationsmaschinerien, fokussieren, vgl. z. B. [SL04], weisen Parallelen zu Testansätze für die eigentlichen *MTs* auf und lassen sich ggf. auch diesbezüglich wiederverwenden [DPV08; Hil+12; WS12].

5. *MBT*-Ansätze, vgl. Abschnitt 6.1, besitzen ebenfalls Berührungspunkte, da einerseits (deklarative) Modelltransformationen als Testmodelle genutzt werden können, vgl. z. B. [BKS04] oder [HL05; HM05], und andererseits *MT*-Programme grundsätzlich auch modellbasiert getestet werden können.

Überdeckung und Adäquatheit

Um Testmengen bewerten oder zielgerichtet verbessern zu können, wird ein Adäquatheitskonzept, z. B. auf Grundlage eines Überdeckungskriteriums, benötigt. Für *MT*-Programme wurden einige Ansätze speziell entwickelt, von denen einige im Folgenden überblicksartig beschrieben werden. Dabei stellt die überwiegende Mehrzahl der entwickelten Kriterien *Black-Box*-Ansätze dar, die rein funktionsbasiert das Ein- und Ausgabeverhalten betrachten.

Metamodell- und Partitionenüberdeckung Da *MTs* als Abbildungen zwischen Metamodellen aufzufassen sind, liegt es nahe, Überdeckungskonzepte im Hinblick auf Metamodelle und deren Elemente zu formulieren.

Fleurey et al. präsentieren in [FSB04] ein Überdeckungskriterium auf Basis sowohl der Überdeckungskriterien für UML-Klassendiagramme aus [And+03] als auch dem Kategorie-Partitionierungsverfahren von Ostrand und Balcer [OB88]. Gefordert wird, dass Eingabemodelle so gewählt werden, dass bestimmte Eigenschaften des Metamodells der

Eingangssprache abgedeckt werden, also z. B. Objektstrukturen bestimmte Assoziationsmultiplizitäten abdecken oder repräsentative Werte aus bestimmten Attributwerteklassen in Modellen vorkommen. Darüber hinaus führen die Autoren das Konzept des *effektiven Metamodells* ein, welches beinhaltet, dass nur die Teile des ursprünglichen Metamodells bei der Überdeckungsbestimmung berücksichtigt werden, die auch tatsächlich eine Relevanz für die Transformation besitzen. Konkret bedeutet dies, dass die berücksichtigten Bestandteile entweder durch *OCL*-Bedingungen oder durch die Implementierung explizit referenziert werden müssen. Baudry et al. erweitert das Überdeckungskonzept später noch um *Objekt-* und *Modellfragmente* zur Beschreibung von Kombinationen zu überdeckender Eigenschaften [Bau+06].

Wang et al. definieren Metamodellüberdeckung für das konkrete *MOF-Meta*metamodell anhand der Überdeckung von sog. *Features* (im Sinne von Properties bzw. Attributen bezogen auf eine bestimmte Klasse) [WKC06]. Zur Überdeckung eines Features muss das entsprechende Feature durch mindestens eine Transformationsregel referenziert werden, was bereits statisch, ohne dynamische Ausführung der Regeln analysiert werden kann. Ausgehend von Feature-Überdeckung werden noch *Inheritance-Coverage* (Feature-Überdeckung für alle geerbten Features einer bestimmten Klasse, die Unterklasse einer anderer Klasse ist), *Association-Coverage* (Feature-Coverage für alle Features der Klasse auf das betrachtete konkrete Assoziationsende zeigt), *Model-Element-Coverage* (Feature-, Inheritance- und Association-Coverage für das entsprechende Element) und *Metamodell-Coverage* (Model-Element-Coverage für alle Elemente) definiert. Ein Traversierungsalgorithmus zur Bestimmung der zu überdeckenden Coverage-Items anhand der Ein- und Ausgabemetamodelle sowie der Transformationsregeln wird skizziert und eine Umsetzung in Tefkat beschrieben.

Stürmer et al. nutzen in [SC03; SC05; Stü+07a] dagegen die Klassifikationsbaum-Methode aus [GG93]. Ziel ist es, die im Rahmen der Methode bestimmten und verwalteten Partitionen – diese werden anhand der *LHSs* von *GT*-Regeln sowie besonderen Attributwerten bestimmt – zu überdecken. Allerdings bleiben Details teilweise im Dunkeln, da nur angedeutet wird, wie der Klassifikationsbaum genau aus einer Regel und dem Metamodell entsteht.

In [Fle+09] stellen Fleurey et al. ein Metamodell zur Beschreibung von Partitionen und Kombinationen von Modell- und Objektfragmenten für das Testen mit Metamodellüberdeckung vor. Die Grundidee besteht darin, so systematische und sinnvolle Kombination von Werten bzw. Wertebereichen der Partitionen für generische Testkriterien unabhängig von konkreten Metamodellen ableiten zu können. Hierzu werden verschiedene Strategien und Testkriterien entwickelt, und diese in einem auf *EMOF* aufbauenden Rahmenwerk, dem *Metamodel Coverage Checker (MMCC)*, realisiert. So ist es möglich, eine gegebene Testmenge in Bezug auf die Metamodellüberdeckung hinsichtlich der Eingabesprache automatisiert zu bewerten.

Constraints und Verträge Cariou et al. schlagen in [Car+04a; Car+04b] vor, den *Design-By-Contract*-Ansatz aus [Mey92] auf die Spezifikation von *MT*-Programmen zu übertragen, z. B. um diese dadurch besser testen zu können. Dabei werden drei Mengen von Verträgen unterschieden: (i) Bedingungen auf Quellseite, (ii) Bedingungen auf Zielseite sowie (iii) Bedingungen, die sich auf die Abbildung selbst beziehen. Die Autoren sprechen sich dafür aus, die Verträge mit *OCL* zu formulieren und beschreiben zwei

Arten, wie diese formuliert werden können, nämlich per „Standard“-Weg über vor und Nachbedingungen einer Operation mit Hilfe des `@pre`-Schlüsselwortes, vgl. hierzu vor allem [Car+04b], oder über einen Ansatz zur Beschreibung von Abbildungen zwischen *UML-Packages*, vgl. [Car+04a].

[Bau+06] unterscheiden dagegen drei Arten von Verträgen, nämlich (i) Verträge im Sinne von Cariou et al. [Car+04a; Car+04b], (ii) Verträge zwischen den Bestandteilen der Transformation, also z. B. zwischen Operationen, sowie (iii) Zusicherungen, die sich nur und ausschließlich auf die Ausgabeseite beschränken und die unabhängig von den Nachbedingungen der Transformation zu sehen sind.

Küster und Abd-El-Razik schlagen in [KA07] vor, dass für jeden (*OCL-*)Constraint mindestens ein Testfall erzeugt wird. So soll ausgeschlossen werden, dass Fehler übersehen werden, die zu einer Verletzung der jeweiligen Bedingung durch die Transformation führen. Dazu werden die durch die Transformation modifizierten Modellelemente betrachtet, um so die Bedingungen zu identifizieren, die durch die Anwendung einer bestimmten Regel potentiell verletzt werden könnten. Anschließend ist die Einhaltung jeder betroffenen Bedingung durch Tests zu überprüfen.

Bauer et al. greifen in [BKE11] einige der Metamodell-Überdeckungskriterien von Fleurey et al. aus [Fle+09] auf und ergänzen sie um *Feature-Coverage* (ein Feature ist hier eine Kombination von Modellelementen, im Beispiel definiert durch ein mittels *OCL* beschriebenes Muster) sowie *Transformation-Contract-Coverage* (pro Vertragsregel ein Coverage-Item). Die beiden neuen Kriterien können somit als Überdeckung von Bedingungen und Verträgen aufgefasst werden. Mit Hilfe dieser Überdeckungskriterien beschreiben die Autoren einen Ansatz zum Testen von verketteten Transformationen (engl. Transformation Chains), der auf sog. *Footprints* basiert. Dabei handelt es sich um Angaben zu den einzelnen erreichten Überdeckungswerten, die zu Vektoren zusammengefasst sind. Diese lassen sich vergleichen und ordnen und ermöglichen so eine Minimierung bzw. Optimierung der Testmenge(n).

TRACTs In [GV11; Val+12] stellen Gogolla und Vallecillo et al. sog. *TRACTs* als eine verallgemeinerte Form von *MT*-Verträgen vor, die als Grundlage zum Testen geeignet sind. *TRACTs* beinhalten jeweils Bedingungen für das Quell- sowie das Zielmetamodell zusätzlich zu den immanenten Bedingungen der Sprachen und Bedingungen für die zu testenden Abbildungen, übertragen auf die Beziehung zwischen den beteiligten Metamodellen. Konkret lassen sich entsprechende Bedingungen in Form von *OCL* ausdrücken. Zusätzlich umfassen *TRACTs* Beschreibungen einer Test-Suite oder Mengen von Testmodellen (Instanzen des Quellmetamodells). Ein einzelner *TRACT* definiert ein konkret zu testendes Szenario der noch zu entwickelnden oder noch zu testenden Transformation; in ihrer Gesamtheit stellt die Menge von *TRACTs* eine deklarative Spezifikation einer *MT* dar [Val+12]. Pro *TRACT* wird mindestens ein Testfall benötigt, welcher sich ggf. auch anhand der Bedingungen für die Eingabeseite automatisiert generieren lässt [GV11] – beispielsweise mit Hilfe des *USE*-Werkzeugs und einer *ASSL*-Beschreibung [GBR05].

Template-Überdeckung Küster und Abd-El-Razik beschreiben in [KA07], neben der oben erwähnten Richtlinie zur Erstellung von Testfällen, zwei weitere Strategien zur Konstruktion von Testeingabemodellen für das Testen von *MT*. Sie schlagen z. B. auch vor, anhand der *LHS* von konzeptuellen Transformationsregeln (unterspezifizierte bzw. skiz-

zenhafte Regeln) sog. parametrisierbare *Meta-Model-Templates* abzuleiten. Aus diesen können dann systematisch, durch kombinatorische Verfahren unter Berücksichtigung der Metamodellüberdeckung, Eingabemodelle generiert werden, um mit diesen die Implementierungen der Regeln zu testen.

White-Box-Ansätze Bisher wurden vorwiegend Überdeckungskriterien für das funktionale Black-Box-Testen anhand von Metamodellen und Bedingungen bzw. Zusicherungen betrachtet. Diese haben den großen Vorteil, unabhängig von der *MT*-Sprache zu sein, können dadurch aber auch nicht auf Eigenheiten einer konkreten Sprache bzw. Implementierung eingehen. Im Folgenden werden Ansätze behandelt, die entweder speziell auf eine bestimmte *MT*-Sprache hin zugeschnitten sind oder aber stark durch eine konkrete Implementierung beeinflusst werden.

Die Arbeit [FSB04] von Fleurey et al. beispielsweise kann insofern als White-Box-Arbeit interpretiert werden, als dass die Bestimmung des effektiven Metamodells auch auf Grundlage von aus der Implementierung abgeleiteten Informationen erfolgt.

Der Ansatz [KA07] von Küster und Abd-El-Razik wird von den Autoren selbst als White-Box-Testansatz bezeichnet. Vermutlich, weil er einerseits speziell für *MTs* im Kontext der Business-Driven-Development entwickelt wurde, andererseits, da er direkt auf den (konzeptuellen) Regeln aufbaut. Allerdings erfolgt die eigentliche Implementierung der Transformation manuell mittels Java, so dass es sich bei den Regeln eher um eine Spezifikation bzw. ein Testmodell als um eine Implementierung handelt. Dieser Interpretation nach wäre der Ansatz ggf. auch als Black-Box-Ansatz aufzufassen. Die Forderung nach Tests zur Überdeckung von Regelpaaren kann allerdings als eine Form der White-Box-Überdeckung gesehen werden.

Ciancone et al. stellen in [CFM10; CFM13] ihren *MANTra* (Model trANsformation Testing) genannte (White-Box) Unit-Testing-Ansatz für die standardisierte *QVTO*-Sprache vor. Die Tests sowie die auf Zusicherungen beruhenden Orakel werden selbst in QVT spezifiziert. Allerdings fehlen Hinweise auf konkrete Überdeckungskriterien zur zielgerichteten Ableitung von Tests („Input Domain Coverage“ wird erwähnt). Dagegen wird auf Domänenexperten und auf zufallsbasierte Testgeneratoren als Quelle für Tests verwiesen.

McQuillan und Power präsentieren in [MP09] einen White-Box-Überdeckungsansatz für *ATL*-Transformationen. Technisch basiert er auf der Analyse der Eingabe für die Virtuelle Maschine von *ATL* sowie auf den im Rahmen einer Ausführung gesammelten Debug-Informationen. Ausgehend davon werden drei Überdeckungskriterien vorgeschlagen, nämlich (i) Regelüberdeckung (Anteil der aufgerufenen ATL-Regeln), (ii) Anweisungsüberdeckung (Anteil der im Kompilat vorhandenen und ausgeführten Byte-Code-Anweisungen bezogen auf alle vorhandenen Anweisungen) sowie (iii) Entscheidungsüberdeckung (Pfadüberdeckung bezüglich der Verzweigungen bei `IF`-Anweisungen), wobei der Einsatz der letztgenannten Variante empfohlen wird.

Einen weiteren White-Box-Ansatz für *ATL* beschreiben Gonzalez und Cabot in [GC12]. Ausgehend von einer *ATL*-Beschreibung werden *OCL*-Ausdrücke abgeleitet. Diesen werden dann jeweils Knoten in einem sog. Abhängigkeitsgraphen zugeordnet; die Kanten ergeben sich anhand von Abhängigkeiten zwischen den Ausdrücken. Unter Beachtung verschiedener Kriterien, die über eine Form der Bedingungsüberdeckung definiert sind, werden im Anschluss Teile der Graphen überdeckt. Die bei der Überdeckung berücksich-

tigten Ausdrücke lassen sich dann zur Generierung von Testmodellen mittels *EMFtoCSP* [Gon+12] zusammenfassen und verwenden.

In [Sch+13] stellen Schönböck et al. einen als *generisch* beschriebenen White-Box-Ansatz zum Testen auf Grundlage des sog. *Symbolischen Ausführens* [Cla76] vor. Als generisch gilt der Ansatz deshalb, weil er nicht von einer bestimmten Transformationssprache abhängt. Vorausgesetzt wird dagegen einen Abbildung zwischen der *MT*-Sprache und der im Ansatz zum Einsatz kommenden Kontrollflussgraphen-Sprache. Mittels des *UMLtoCSP*-Werkzeugs [CCR07] werden auch hier Testmodelle generiert, und zwar anhand von Bedingungen, die sich durch Traversierung des Kontrollflussgraphen bestimmen lassen, den symbolischen sog. *Pfad-Constraints*.

Mutationsanalyse und Mutationsbasiertes Testen Neben den üblichen Überdeckungskriterien existieren auch im Zusammenhang mit *MTs* mutationsbasierte Ansätze [DLS78]. Sehr bekannte Papiere in diesem Kontext sind z. B. [MBT06a; MBT06b] von Mottu et al. Es existieren aber beispielsweise mit den Arbeiten von Aranega et al. [Ara+10] sowie Khan und Hassine [KH13] weitere Vertreter.

Die Effektivität dieser Verfahren hängt stark von der Qualität der Mutationsoperatoren ab, wie beispielsweise in [AO08, S. 173-174] ausgeführt, weshalb eine ausreichend große und vielfältige Menge an unterschiedlichen konkreten Mutatoren wünschenswert erscheint. In der Literatur wurden sowohl generisch anwendbare [MBT06a] als auch sprachspezifische Mutationen – in [KH13] beispielsweise für *ATL* – vorgestellt. Günstig ist, wenn *typische Fehler* durch die Operatoren simuliert werden, was voraussetzt, dass vor der Entwicklung von Mutatoren erst einmal die Fehlerarten dahingehend analysiert werden. In der Literatur werden *Fehlermodelle* beispielsweise für allgemeine *MTs* [MBT06a; Wim+09], für konzeptuelle Regeln [KA07] aber auch für *GTs* [DPV08] vorgestellt.

Wie in Abschnitt 5.4 dargelegt, eignet sich mutationsbasiertes Testen nicht nur zur Bewertung und Verbesserung einer Testmenge, sondern kann auch dazu genutzt werden, um die Leistungsfähigkeit anderer Testverfahren und Überdeckungskriterien abzuschätzen. Beispiele für eine solche Verwendung im *MT*-Kontext sind unter anderem die Arbeiten [Mot+12; GS13; WAS14].

Generierung von Testmodellen

Nach der Betrachtung verschiedener Optionen für Adäquatheits- und Überdeckungsbegriffen für das Testen, stellt sich auch die Frage, wie entsprechende Tests erzeugt werden können. Grundsätzlich ist das Entwickeln von Testmodellen per Hand immer möglich, aber häufig auch mühsam, fehleranfällig und vergleichsweise teuer. In der Literatur werden verschiedene Optionen zur automatisierten Erzeugung diskutiert, von denen einige bis hierhin bereits am Rande erwähnt wurden, die im Folgenden nochmals zusammenfassend vorgestellt werden:

1. *Generieren und Überprüfen.* Eine offensichtliche, ggf. aber beschränkt zielführende Art der Testmodellgenerierung liegt in der Nutzung modellspezifischer oder generischer Algorithmen zur direkten Erzeugung von Instanzen aus einem Metamodell, vgl. z. B. [Bro+06]. Zusätzlich vorhandene Einschränkungen für die zu generierenden Testmodelle – Nebenbedingungen des Metamodells, Vorbedingungen der zu testenden Transformation(en) etc. – lassen sich nach der Generierung überprüfen [Bro+06], so dass invalide Repräsentanten aussortiert werden können.

2. *Kombinatorische Verfahren.* Werden Eingabemodelle mit bestimmten Eigenschaften benötigt, können auch allgemeine Such- bzw. Optimierungsstrategien verwendet werden. In [FSB04] wird beispielsweise ein *genetischer Algorithmus*[21] vorgestellt, der, ausgehend von einer initialen Testmenge, neue Tests durch Rekombinieren und Filtern erzeugt. Der Algorithmus stoppt, sobald alle Coverage-Items, welche anhand der zuvor beschriebenen Metamodellüberdeckung entwickelt wurden, durch die Testmenge abgedeckt sind (Anm.: vermutlich zusätzlich abgesichert durch einen Timer oder eine maximale Anzahl an Versuchen). Wang et al. beschreiben in [WKC08] einen Ansatz, der auf der Bildung von Partitionen, repräsentativer Werte, der Kombination solcher Werte sowie der anschließenden Ableitung konkreter Metamodellinstanzen, welche die ausgewählten Werte nutzen, basiert. Guerra beschreibt in [Gue12] die Nutzung kombinatorischer Verfahren, wie der *T-Wise*-Überdeckung [WP01], zur Generierung von Testmodellen unter Zuhilfenahme von Model-Finding-Techniken.

3. *Model-Finding.* Lässt sich ein gesuchtes Testmodell durch eine Menge von formalisierbaren Aussagen beschreiben, beispielsweise durch eine Menge von *OCL*-Ausdrücken als Ergänzung zu einem Metamodell, so lassen sich Model-Finding-Ansätze zur Erzeugung von Instanzen bzw. Modellen nutzen, vgl. hierzu auch die Aussagen aus Abschnitt 6.2.3. Beispiele für entsprechende Arbeiten sind u. a. [SBM09; Cab+10a; Gue12; GS13].

4. *Grammatik-basiert.* Ein grundlegend anderer Ansatz zur Ableitung von Testmodellen wird in den Arbeiten [Win+08; EKT09; Tae12; HM11; HM10; FMM13] verfolgt. Hierbei wird ein Metamodell bzw. ein Klassendiagramm, das zwar eine Sprache deklarativ beschreibt, aber ohne Weiteres keine operative Darstellung zur Wort- bzw. Instanzengenerierung beschreibt, in eine *Graphgrammatik* übersetzt. Die Menge der Graphgrammatikproduktionen beschreibt ebenfalls eine Sprache, die im Idealfall der Sprache des Metamodells inklusive eventuell vorhandener Einschränkungen durch Invarianten entspricht. Die genannten Quellen beschreiben Ansätze, in denen Konstruktionsvorschriften für solche Graphgrammatiken vorgestellt werden. Im Vergleich zu Model-Finding-Ansätzen ist diese Herangehensweise deshalb vorteilhaft, da sich mit Hilfe von Graphgrammatiken auch sehr umfangreiche Modelle mit vielen Elementen [WAS14] oder sehr viele Instanzen [EKT09] effektiv und effizient erzeugen lassen.

   Ehrig et al. beschreiben in [EKT09] erstmals die automatisierte Ableitung einer Graphgrammatik aus einem Metamodell ohne Nebenbedingungen. In [Tae12] erweitert Taentzer den Ansatz dahingehend, dass Einschränkungen bzgl. der Multiplizitäten von Assoziationsenden durch die Grammatik berücksichtigt werden. Durch die Umwandlung von eingeschränkten *OCL*-Ausdrücken in Anwendungsbedingungen von Regeln, vgl. [Win+08], lassen sich grundsätzlich auch *OCL*-Bedingungen berücksichtigen. Hoffmann und Minas benutzen in [HM11; HM10] sog. *Adaptive Star Grammars* nach [Dre+06] und sind dadurch nicht auf *NACs* oder unterschiedliche Anwendungsphasen verschiedener Regelgruppen angewiesen. Fuerst et al. [FMM13] stellen ebenfalls einen grammatikbasierten Ansatz vor, nutzen dabei allerdings sog. *Layered Graph Grammars (LGGs)* nach [RS97]. Darüber hinaus

[21] Von den Autoren als *Bacteriologic Algorithm* bezeichnet.

schlagen Fuerst et al. vor, die so entwickelte Grammatik auch zur semantischen Analyse von zugehörigen Instanzmodellen zu nutzen.

Neben den Graphgrammatiken, die sich ausschließlich auf ein Metamodell beziehen, lassen sich auch *TGGs* [Sch95] zur Generierung von Testmodellen sowie ggf. auch den erwarteten Ergebnismodellen nutzen, wie beispielsweise in [Hil+12; HLG13; WAS14] untersucht.

Orakel

Um das Ergebnis einer Modelltransformation im Hinblick auf Validität bzw. Korrektheit zu bewerten, gibt es verschiedene Möglichkeiten. Mottu et al. identifizieren in [MBL08] drei grundlegende technische Basisoperationen zur Realisierung einer Orakelfunktion, nämlich (i) Modellvergleich, (ii) Auswertung von Verträgen bzw. Zusicherungen sowie (iii) Mustersuche. Aufbauend darauf skizzieren die Autoren sechs konkrete Orakelfunktionen:

1. Nutzung einer Referenzimplementierung im Zusammenspiel mit einem Modellvergleich (zum Vergleich der Ausgaben beider Implementierungen).
2. Anwendung einer inversen Transformation auf die Testausgabe des *SUT* und Vergleich zwischen der Eingabe des *SUT* und dem Resultat der inversen Transformation.
3. Vergleich mit dem erwarteten Ergebnis. Dieses kann beispielsweise händisch konstruiert oder anderweitig abgeleitet sein.
4. Ein (parametrisierter) Vertrag, der Ein- und Ausgabe miteinander in Beziehung setzt und zu `wahr` oder `falsch` evaluiert.
5. (*OCL*-)Zusicherungen, die sich alleine auf die Ausgabeseite beziehen und alle erfüllt sein müssen.
6. Ein Orakel auf Grundlage einer Überprüfung der erwarteten Anzahl von Repräsentanten bestimmter Modellfragmente bzw. der Anzahl von Treffern für bestimmte Muster.

Auch sind Kombinationen von mehreren Teilorakeln möglich, wie das Beispiel [Bau+06] durch die Verwendung sog. *Two-Layer Oracles* zeigt, bei der ein grobes (unempfindliches) und ein feines (sensibles) Orakel miteinander kombiniert werden. Nach [AW13, Kap. 2.1.2] sind in der Literatur Orakel auf Basis von Modellvergleichen [LZG04] oder von Bedingungen am weitesten verbreitet.

Modellvergleich Obwohl sich Modelle in der Regel in Form von *XML* bzw. *XMI* serialisieren lassen, sind Vergleichsalgorithmen für textuelle Sprachen (oder für baumartige Strukturen) als Orakel nur eingeschränkt brauchbar [FW07]. Gründe hierfür liegen in der Nichteindeutigkeit einer solchen Serialisierung, der Sensibilität gegenüber Änderungen in der textuellen Ausgabe, ohne strukturelle oder relevante Änderungen im Modell, z. B. durch sich ändernde intern verwendete Bezeichner, vgl. z. B. [FW07], sowie das oft nur implizite Vorhandensein relevanter Informationen. Zu diesem letzten Punkt vgl. auch [KPP06].

Lin et al. erkennen früh die Bedeutung des Modellvergleichproblems unter anderem für das Testen von *MTs* [LZG04]. In [LZG05] stellen dieselben Autoren ein Testrahmenwerk zum Testen von Modelltransformationen im Kontext des Modellierungswerkzeugs

Generic Modeling Environment (GME) [Léd+01] vor, das einen einfachen Modellvergleich auf Basis von eindeutigen Bezeichnern, engl. Unique Identifiers (UIDs), als Orakel nutzt. Hierbei werden Instanzen eines einzigen Metamodells miteinander verglichen. Andere frühe Arbeiten für (Meta-)Modelle, auf die sich Lin et al. auch jeweils beziehen und die ebenfalls auf Grundlage eindeutiger Bezeichner arbeiten, sind [AP03; OWK03]. Der Fokus dieser Arbeiten liegt allerdings primär auf der Modellversionierung oder der Visualisierung von Modelldeltas.

Kolovos et al. greifen in [KPP06] die Argumentation von Lin et al. auf, merken aber an, dass ein Modellvergleich von Modellen unterschiedlicher Metamodelle und sogar unterschiedlicher Modellierungswelten möglich sein sollte. Hierzu stellen sie eine regelbasierte, textuelle Vergleichssprache, die sog. *Epsilon Merging Language (EML)*[22] vor, mit der sich vom Anwender spezifizieren lässt, welche Elemente eines Modells wie auf Elemente eines anderen Modells abzubilden sind, um als *gleich* im Sinne des Diffs zu gelten. Es wird also die Entwicklung von problemangepassten Modellvergleichen propagiert.

Bekannte und häufig genutzte Algorithmen und Werkzeuge zum Vergleichen von Modellen auf Basis von heuristischen Verfahren auf Basis der Ähnlichkeit von Modellelementen sind unter anderem *UMLDiff* [XS05], *EMFCompare* [BP08] und, insbesondere für große Modelle geeignet, *SiDiff*[23] [Tre+07; KWN05]. Solche Ansätze sind allerdings nicht immer optimal, wie Taentzer et al. in [Tae+14] für *EMFCompare* bemerken.[24]

Es existieren mit [FW07; Kol+09] auch mindestens zwei Übersichtsarbeiten zu dem Themenkomplex Modellvergleich. Beide unterscheiden sich hinsichtlich ihrer inhaltlichen Ausrichtung: Förtsch und Westfechtel untersuchen in [FW07] neben Ansätzen zur Berechnung von Modellunterschieden auch Arbeiten zum Verschmelzen von Modellen. Kolovos et al. betrachten in [Kol+09] dagegen ausschließlich den Aspekt des Modellvergleichs, mit starkem Fokus auf den technischen Herausforderungen konkreter Umsetzungen.

Neben syntaktischen Unterschieden können auch *semantische* Unterschiede bei einem Modellvergleich zugrunde gelegt werden [MRR11a; LMK14] und somit als Grundlage für ein Orakel dienen. Mag dies bei Modellen mit einer *dynamischen* Ausführungssemantik noch unmittelbar ersichtlich sein – vgl. hierzu die bereits erwähnte Arbeit von Varró und Pataricza [VP03] – so existieren auch Arbeiten, die ohne Ausführungssemantik auskommen. Maoz et al. verfolgen in [MRR11c] einen Ansatz, bei dem sie semantische Unterschiede für *statische* Modelle – konkret Klassendiagramme – mit Hilfe der bereits in Abschnitt 6.2.2 erwähnten *Diff-Witnesses* untersuchen. Dabei handelt es sich in diesem speziellen Fall um Objektdiagramme, die insofern auf Unterschiede in der Semantik hindeuten, als dass sie für ein Diagramm eine valide Instanz darstellen, für das andere aber nicht.

Langer et al. nutzt dagegen die zuvor bereits erwähnte *Epsilon Comparison Language (ECL)*, um so, und das ist bemerkenswert, semantische Vergleiche über aufgenommenen Traces durchzuführen. Entsprechende Traces werden im Rahmen der Ausführung der zu vergleichenden Modelle anhand konkreter Eingaben bestimmt. Die Eingaben können durch Constraint-Solving (konkret mit dem Ansatz von Kuhlmann et al. [KHG11]) von Pfadbedingungen generiert werden. Letztere lassen sich wiederum durch symbolische Ausführung so bestimmen, dass alle möglichen bzw. relevanten Ausführungspfade

[22] Später zu einer separaten Sprache, der *Epsilon Comparison Language (ECL)*, für den reinen Vergleich von Modellen (ohne Bezug zur Verschmelzung) weiterentwickelt., vgl. [Kol+14, Kap. 8].
[23] `http://sidiff.org/` (zuletzt abgerufen am 7.10.2014).
[24] Die auftretenden Schwierigkeiten ließen sich in diesem Fall aber durch Anpassungen korrigieren.

abgedeckt werden. Die vorgestellte Methodik ist generisch, und zwar in dem Sinne, dass sie unabhängig von konkreten Modellsemantiken ist.

Weitere wichtige Orakel-Optionen Nachdem wichtige Orakeloptionen bereits erwähnt wurden und auf einige herausragende Arbeiten zu Orakeln auf Basis eines Modellvergleichs eingegangen wurde, werden nun noch einige Hinweise auf Literatur für alternative Ansätze gegeben. Diese Alternativen basieren beispielsweise auf den folgenden Konzepten:

1. Verträge [MBT06b; LBJ06; Car+09],
2. Graphische Verträge, engl. Visual Contracts, [HL07; HKM11; KRH12a; KRH12b; Gue+13],
3. Graphmuster, wie z. B. in [GZ05], wo aus Graphmustern JUnit-Assertions generiert werden,
4. Alternative Implementierungen, z. B. in Form einer älteren Version (z. B. Prototyp) der Implementierung (vgl. hierzu die Konzepte Regressions- und Back-to-Back-Testen [Lig02]), eines Interpreters [SC03; DPV08; Hil+12] oder der Operationalisierung von Transformationsmodellen [Cab+10c; Büt+12b],
5. Testszenarien; das Orakel bewertet das ausführbare Ergebnis einer *MT* durch Testen dieser Ausgabe mit Hilfe der Testszenarien wie z. B. in [Bau+06; Stü+07a].

Testen von M2T-Transformationen

Obwohl oder gerade weil das Testen von *M2T*-Transformationen in vielen Fällen Ähnlichkeiten zum viel erforschten Compiler-Testing[25] aufweist, ist die Anzahl entsprechender Arbeiten, die sich dediziert mit diesem Teilaspekt des *MT*-Testthemenkomplexes auseinandersetzen, vergleichsweise gering, wie auch von Wimmer und Burgueño in [WB13] festgestellt. Über mögliche Gründe lässt sich höchstens spekulieren. Einerseits lassen sich bestehende Testansätze für Compiler oder *M2M*-Transformationen zum Teil direkt wiederverwenden, insbesondere, da Teilschritte oft als *M2M*-Abbildung beschreibbar sind, vgl. z. B. [WB13]. Andererseits werden *M2M*-Abbildungen von Forschern, die sich mit *MTs* (oder *GTs*) beschäftigen, generell häufiger berücksichtigt.

Stürmer und Conrad [SC03; SC05; Stü+07a] stellen ein Verfahren zum Testen von Codegeneratoren vor, die im Rahmen der modellbasierten Entwicklung Eingebetteter Systeme zum Einsatz kommen. Konkret werden Codegeneratoren für Matlab/Simulink (bzw. *TargetLink* von dSPACE) getestet. Dazu werden notwendige Optimierungsschritte des Codegenerators mittels *GT*-Regeln formalisiert, anschließend in eine Klassifikationsbaumdarstellung übertragen und die daraus ableitbaren abstrakten Testfälle in Testmodelle übersetzt (sog. *First-Order-Tests*). Nachdem der Codegenerator die Testmodelle in ausführbaren Code übersetzt hat, werden Testvektoren (sog. *Second-Order-Tests*) genutzt, um den resultierenden Code zu testen. Als Orakel dient dabei eine problemspezifische Vergleichskomponente, welche die Ausgaben des Codes mit denen des Simulink-Simulators, welcher die Testmodelle auf Grundlage der Testvektoren ebenfalls ausführen kann, vergleicht.

[25] Für eine Übersicht siehe [KP05]

Baldan et al. greifen die Idee der Spezifikation von Optimierungsschritten eines Compilers mit Hilfe von *GT*-Regeln in [BKS04] ebenfalls auf. Mit Hilfe zweier Graphgrammatiken, einer *generierenden Grammatik* und einer *optimierenden Grammatik* beschreiben sie einerseits die Menge der möglichen Eingaben und andererseits die zu untersuchenden Optimierungsschritte. Für beide Grammatiken identifizieren die Autoren anhand ihres Unfolding-Ansatzes Mengen von für das Testen interessanter Regelanwendungssequenzen. Diese lassen sich anschließend durch isolierte Betrachtung der Anwendungen der generierenden Anteile der Regeln in Testmodelle übersetzen.

Einen weiteren Ansatz zum Testen von Optimierungsschritten eines Codegenerators, genannt *Graph Optimizer Testing Kit (GraphOTK)*, stellen Zelenov et al. in [Zel+06] vor. Anhand der *LHSs* von identifizierten Optimierungsregeln, die sich zum Teil auf abstrakte Klassen beziehen, werden durch kombinatorische, problemspezifische Ad-hoc-Generierungsalgorithmen Testmodelle erzeugt. Konzeptuell ähnelt der Ansatz damit in Teilen der Arbeit von Küster und Abd-El-Razik [KA07], da auch dort unter anderem konkrete Tests aus der *LHS* von Regeln abgeleitet werden.

Wimmer und Burgueno [WB13] beschreiben, wie sich *M2T*- sowie dazu inverse *T2M*-Transformationen mittels der zuvor bereits erwähnten *TRACTs* testen lassen. Dazu nutzen sie ein generisches Metamodell zur Beschreibung von textuellen Artefakten mittels Konzepten wie `File`, `Folder` und `Line`, und fassen die zu testenden Transformationen als *M2M*-Abbildungen zwischen diesem Metamodell und anderen Metamodellen auf. Mit Hilfe des vorgestellten Ansatzes testen die Autoren sechs Codegeneratoren, die aus *UML*-Klassendiagrammen Java-Code erzeugen.

Graphtransformationen und Testen

Die meisten Testverfahren für *MTs* sind als unabhängig von der konkreten Transformationssprache anzusehen und damit grundsätzlich auch für *GTs* geeignet. Darüber hinaus gibt es allerdings auch *GT*-spezifische Ansätze, von denen einige in diesem Abschnitt vorgestellt werden. Im Folgenden werden zwei Aspekte betrachtet, nämlich (i) Testansätze speziell für *GTs* sowie (ii) Testansätze für *MTs*,[26] die sich im wesentlichen Maße auf *GTs* und deren (formale) Eigenschaften stützen.

Testen von *GTs* Darabos et al. stellen beispielsweise in [DPV08; Dar07] einen Ansatz zum Testen von Implementierungen einzelner *GT*-Regeln vor. Ziel ist es, Fehler in einem Codegenerator für die *GT*-Regeln oder Fehler einer manuellen Implementierung zu erkennen.

Das dazu vorgestellte Überdeckungskriterium ist über eine spezielle Art der Abdeckung von Booleschen Formeln definiert. Die sich ergebenden Bitvektoren steuern die Anwendung von Mutationsoperatoren, welche wiederum – durch Anwendung auf die *LHSs* der Regeln und der Interpretation der Muster als Objektdiagramme – zu partiellen Testmodellen führen. Es wird vorgeschlagen, als Orakel einen Vergleich mit einer Referenzimplementierung in Form eines Interpreters zu nutzen.

Guerra et al. bieten mit der Pamomo-Sprache (Pattern-based Modeling Language for Model Transformations) und dem PaCo-Werkzeug (Pamomo Contract-Checker), vgl. z. B. [Gue+13], einen Ansatz zum spezifikationsbasierten Testen von bidirektionalen

[26] Diese sind von den bereits ab S. 119 beschriebenen formalen Verifikationstechniken zu unterscheiden.

MTs, insbesondere auch von *TGGs*. Bei diesem Ansatz werden Verträge formuliert, aus denen partielle Orakel und Testmodelle generiert werden können [Gue+13; GS13; Gue12]. Die Orakel basieren technisch auf einer QVT-Implementierung. Testmodelle werden mittels Constraint- bzw. SAT-Solver erzeugt, vgl. hierzu die zuvor erwähnten Arbeiten zum Thema *Model-Finding*.

In seiner Dissertation [Gei11] beschreibt Geiger unter anderem Überdeckungskonzepte für *Fujaba Story-Diagramme*, um damit Testszenarien für das szenariobasierte Testen zu entwickeln. Bezogen auf dieses Ziel, hat der entsprechende Teil von Geigers Arbeit gewisse Ähnlichkeit zu einem der Hauptziele der hier vorliegenden Arbeit, vgl. insbesondere Kapitel 7. Was die Ausgangslage betrifft, weist Geigers Arbeit folglich von allen Überdeckungsansätzen für *MTs* oder *GTs* die größte Nähe zu der vorliegenden Arbeit auf. Allerdings werden die beiden vorgeschlagenen Überdeckungskriterien für Story-Diagramme – (a) *(extended) Activity-Coverage ((e)AC)* sowie (b) *Search Coverage (SC)* – sowie eine Form der Überdeckung von Klassendiagrammen nur kurz skizziert, vgl. [Gei11, S. 74 f.]. Activity-Coverage fasst Activity-Knoten als atomare Blöcke, sprich Knoten eines Kontrollflussgraphen auf, und überträgt die klassischen Überdeckungskonzepte Anweisungs-, Verzweigungs-, Pfad- und (vollständige) Bedingungsüberdeckung auf diese. Die *erweiterte* Anweisungsüberdeckung im Sinne von *eAC* soll sicherstellen, dass für jeden Activity-Knoten der Fall auftritt, dass das Muster erfolgreich ausgeführt wird und auch fehlschlägt. Der Unterschied zur Verzweigungsüberdeckung bleibt dabei allerdings unklar. Der Search Coverage werden atomare Basisoperationen der Graphmustersuche zugrunde gelegt, die auch zur Beschreibung des Suchplans genutzt werden können, und die im Rahmen der Codegenerierung von Fujaba ihre Entsprechung in zugehörigen Codeabschnitten finden. *SC0* bedeutet, dass „jede Operation zur Graphsuche mindestens einmal ausgeführt wird“ [Gei11, S. 74], für das komplette Muster also ein Treffer gefunden wird. *SC1* verlangt, dass „jede Operation einmal erfolgreich und einmal nicht erfolgreich durchgeführt wird“ [Gei11, S. 74]. Die Beschreibungen der beiden übrigen Search-Coverage-Kriterien *SC2* und *SC3* lassen Raum für Interpretation. *SC2* bezieht sich explizit auf „optionale Objekte“ – vermutlich also optionale Knoten innerhalb eines Musters – und fordert „jede Kombination von ‚wird gefunden‘und ‚wird nicht gefunden‘“ [Gei11, S. 74]. *SC3* bezieht sich auf „Constraints“. Vermutlich sind damit Attributbedingungen gemeint. In [Gei11] wird für diese gefordert, „sämtliche Evaluierungsmöglichkeiten [zu] berücksichtigen“. Die vorgeschlagene, spezielle Form der Klassendiagrammabdeckung fordert, dass

> „• jede Blattklasse instanziiert und ein Objekt der Klasse wieder gelöscht wird.
>
> • jede Assoziation abgedeckt ist (also angelegt, traversiert und gelöscht wird).
>
> • jedes Attribut abgedeckt ist (lesender und schreibender Zugriff).
>
> • jede Methode (inklusive der überschriebenen) aufgerufen wird“ [Gei11, S. 75].

Interessant ist aber vor allem die vorgeschlagene Art der technischen Realisierung: es wird die Anweisungsüberdeckung auf Quellcodeebene bestimmt und diese auf die Überdeckung auf Modellebene zurückgerechnet. Dies wird durch eine Anpassung des Codegenerators ermöglicht, der zusätzliche Informationen zur Rückwärtsnavigation in

das Generat einbringt. Motiviert wird die Validität dieses Vorgehens durch die von Baresel et al. in [Bar+03] festgestellte Korrelation zwischen Code und Modellüberdeckung. Allerdings wurde dies nur für den speziellen Fall von Simulink- bzw. Stateflow-Modellen und C-Code exemplarisch gezeigt. Darüber hinaus deuten Ergebnisse von Hildebrandt et al. aus [Hil+12] an, dass die Codeüberdeckung im Falle Interpreter-basierter Ansätze nur sehr eingeschränkte Aussagekraft für das Testen von *GTs* besitzt.

Einen Ansatz zum Testen der Operationalisierung von *TGG*-Regeln, und damit indirekt auch entsprechender Compiler, stellen Hildebrandt et al. in [Hil+12] vor. Kernpunkt ist die Interpretation der Regeln einer *TGG*-Spezifikation als Produktionen für den synchronen Aufbau von Modelltripeln, bestehend aus einer Instanz der „linken“ Sprache, der „rechten“ Sprache sowie des Korrespondenzmetamodells. Die Instanzen der linken (bzw. rechten) Sprache dienen dann als Eingabe für die Operationalisierung der Vorwärtstransformation (bzw. der Rückwärtstransformation) und die zugehörige Instanz auf der anderen Seite des Tripels repräsentiert das erwartete Ergebnis. Letzteres ist aufgrund der Einschränkungen bzgl. der eingesetzten *TGG*-Variante eindeutig. In [HLG13] stellen Hildebrandt et al. darüber hinaus ein erweitertes Überdeckungskonzept mit Bezug zur Interaktion von Regeln als Stoppkriterium im Rahmen der Testableitung vor. Der Ansatz weist somit gewisse Ähnlichkeiten zu Arbeiten auf, die sich mit dem Testen auf Grundlage von String-Grammatiken [Mau90] bzw. dazu passender Überdeckungskriterien [Pur72; Läm01; LS06] auseinandersetzen.

Testen mit *GTs* In [GZ05] stellen Geiger und Zündorf einen szenariobasierten Testansatz namens *Story Driven Testing* vor. Dabei werden Use-Cases mittels Story-Boarding-Techniken in *Fujaba* beschrieben, um daraus u. a. Testfälle und konkrete JUnit-Tests abzuleiten. Aus einfachen Graphmustern werden Testmodelle und Orakel generiert, indem einerseits Muster als Objektdiagramme interpretiert werden und andererseits Testausgaben gegebene Muster enthalten müssen.

Die bereits mehrfach erwähnten Arbeiten von Stürmer et al. [SC03; BKS04; Stü+07a] repräsentieren ebenfalls Beispiele für Arbeiten, in denen ein System – im konkreten Fall optimierende Codegeneratoren – mittels Testverfahren für Graphtransformationen getestet werden. Testmodelle sind dabei einerseits die *GT*-Regelrepräsentation der Optimierungsschritte sowie die Klassifikationsbäume zur Erzeugung der abstrakten Testfälle.

Ähnlich sind auch die drei Strategien zur Testerzeugung von Küster und Abd-El-Razik [KA07] zu sehen. Auch hier bilden *GT*-Regeln sowie deren Interaktion die Grundlage des modellbasierten Testansatzes für die manuell implementierten Transformationen für *BPMN*-Artefakte.

Heckel et al. nutzen ebenfalls *GT*-Regeln, um auf dieser Basis Web-Services zu testen [HM05; HL07; HKM11]. In [HKM11] wird dabei ein Überdeckungskriterium für *GT*-Regeln auf Basis der beiden Relationen *Regelabhängigkeit* und *Regelkonflikt* über der Abdeckung der Kanten eines Abhängigkeitsgraphen definiert. Die Form der Abdeckung weist gewisse Parallelen zur klassischen Datenflussüberdeckung auf, benutzt allerdings auch keinen Kontrollflussgraphen.

In einer eigenen Arbeit, [WAS14], habe meine Kollegen und ich den Einsatz von *TGGs* beim Testen von Modelltransformationen unter praktischen Gesichtspunkten untersucht. Auch wurde der Aspekt nichtdeterministischer Übersetzungen betrachtet, und verschiedene Orakeloptionen anhand einer Beispieltransformation verglichen. Dabei standen ins-

besondere die beiden aus *TGG*-Operationalisierungen ableitbaren Optionen, Modellvergleich, anhand von generierbaren Modellpaaren, sowie Korrespondenzmodellkonstruktion (i. S. v. der Suche nach einem Ableitungspfad zur Konstruktion eines konsistenten Modelltripels) im Blickpunkt der Arbeit.

6.4 Zusammenfassung und Bewertung

Hauptmotivation der Arbeit ist der Wunsch nach einem praktikablen Verifikationsansatz für programmierte Graphtransformationen. Für dieses Ziel sind Arbeiten zu *formalen* Verifikationstechniken von untergeordnetem Interesse. Konstruktive sowie statisch analysierende Verfahren zur Qualitätssicherung berücksichtigen die dynamischen Aspekte von Transformationen nicht ausreichend. Dagegen sind Arbeiten zu *testenden* Verfahren für Modell- und Graphtransformationen unmittelbar relevant.

Alle drei großen Herausforderungen beim Testen von Modelltransformationen nach [Bau+06; Bau+10] wie (i) die möglichst automatische Erzeugung von Testmodellen, (ii) die Bewertung von Tests, als Hilfsmittel bei der Konstruktion weiterer Tests oder zur Etablierung eines Stoppkriteriums, sowie (iii) die Wahl eines geeigneten Orakels wurden in der Literatur bereits untersucht und wichtige Vertreter in diesem Kapitel vorgestellt.

Die Möglichkeiten zur Entwicklung eines *generischen* Orakels sind grundsätzlich einschränkt, wie die starke Fokussierung auf generische Modellvergleichsalgorithmen zeigt. Problemübergreifend einsetzbare Techniken, auf denen ein Orakel seine Entscheidung ableiten kann, sind verfügbar, müssen allerdings i. d. R. an die konkrete Aufgabe adaptiert werden. Problemspezifische Orakel sind dagegen so vielfältig, wie die Transformationsaufgaben und die beteiligten Sprachen selbst. Das Orakelproblem erscheint aus Forschungssicht somit zurzeit weniger interessant, da verallgemeinerbare Ansätze bereits ausgiebig betrachtet wurden. Die praktische Bedeutung des Problems ist allerdings groß.

Die Testdatengenerierung ist dagegen ein äußerst spannendes Feld und wurde bereits mit Hilfe vielfältiger Herangehensweisen untersucht. Hier besteht weiterhin Forschungs- und Optimierungsbedarf. Allerdings setzen optimierte bzw. optimierende Testgenerierungsansätze ein geeignetes Konzept zur Bewertung von Testmengen voraus. Da dieses dritte Problemfeld auch noch nicht als abgeschlossen angesehen werden kann, spricht die zentrale Bedeutung für eine Konzentration auf diesen Aspekt.

Für das Ziel der Entwicklung einer entsprechenden Bewertungsgrundlage in Form eines Überdeckgungskonzeptes sind grundsätzlich alle Arbeiten interessant, die sich mit Fragen der Testüberdeckung auseinandersetzen. Aufgrund der großen Vielfalt und der fehlenden Standardisierung von *MT*-Sprachen haben sich hauptsächlich Ansätze zum funktionalen Black-Box-Testen herausgebildet. Insbesondere die Arbeiten auf Basis formaler Spezifikationsmodelle (wie z. B. Verträge oder TRACTs) erscheinen ausgereift und mächtig – mächtig genug, um beliebige, insb. auch *GT*-basierte *MTs*, testen zu können. Allerdings ist ihre größte Stärke, die Sprachunabhängigkeit, auch gleichzeitig eine große Schwäche, da keine Informationen aus der Implementierung genutzt werden. Aufgrund von Erfahrungen bei der klassischen Entwicklung von Software lässt sich schließen, dass strukturelle White-Box-Verfahren ebenfalls wichtige Komponenten in einem praktikablen Qualitätssicherungsbaukasten sind, da sie die funktionsbasierten Verfahren gut ergänzen. Entsprechende Arbeiten sind folglich genauer zu analysieren.

6.4.1 Abgrenzung von existierenden Ansätze

Es existieren nur wenige strukturbasierte Überdeckungskriterien für *MTs*. Für die hier betrachtete Art der *MTs* (regelbasierte Graphtransformationen mit programmierter Regelanwendung) scheiden auch Ansätze für populäre Sprachen wie *ATL* [MP09; GC12] aufgrund deren Andersartigkeit aus.

Der Ansatz von Schönböck et al., vgl. [Sch+13], kann als *generischer* White-Box-Ansatz umschrieben werden, da er lediglich einen geeigneten Kontrollflussgraphen voraussetzt. Allerdings ist die Angabe eines passenden Kontrollflusses für die zu testenden *MT* u. U. nicht trivial. Eine entsprechende Abbildung muss für jede Transformationssprache ggf. gesondert entwickelt werden. Dabei bleibt offen, ob für eine teilweise deklarative Beschreibung, wie bei programmierten *GTs*, überhaupt eine solch eindeutige Darstellung möglich und *sinnvoll* ist – ggf. müsste dazu ein beliebiger, aber fester Suchplan für einzelne Muster vorausgesetzt werden, was eine zu starke Einschränkung darstellen würde. Obwohl das vorgestellte Verfahren zur Testfallerzeugung auf Basis einer *symbolischen Ausführung* äußerst reizvoll erscheint, kann der Ansatz bzw. Teile davon nicht direkt wiederverwendet werden.

Heckel und Khan et al. beschreiben in [HKM11] bzw. [KRH12a] für ihren auf *GT*-Regeln aufbauenden Testansatz ein Überdeckungskonzept basierend auf *GT*-Regelpaaren, die in Konflikt oder Abhängigkeit zueinander stehen. Obwohl der Ansatz dediziert *GT*-spezifische Aspekte berücksichtigt, sind die Voraussetzungen nicht vergleichbar mit den hier vorliegenden. Im Gegensatz zu den beiden erwähnten Arbeiten finden wir hier eine Situation vor, in der die *GT*-Regeln nicht gleichberechtigt und frei anwendbar nebeneinander existieren. Eine beliebige Reihenfolge bei der Ausführung ist also ausgeschlossen, da die Anwendung durch den Kontrollfluss gesteuert wird. Damit liegt es auch in der Verantwortung des Entwicklers, Konflikte und Abhängigkeiten zu beachten und sonstige Interaktionen zwischen den Regeln explizit zu betrachten.

Der von Stürmer et al. verfolgte Weg, aus dem Musterteil einer *GT*-Regel und dem Metamodell Testmodelle anhand eines Klassifikationsbaumes abzuleiten, vgl. [Stü+07a], erscheint grundsätzlich auch für das Testen einzelner *SDM*-Regeln interessant zu sein, da sich anhand des Klassifikationsbaumes kombinatorische Überdeckungskriterien anwenden ließen. Leider war es mir nicht möglich, die Konstruktion des Klassifikationsbaumes sowie dessen Struktur anhand der verfügbaren Informationen nachzuvollziehen. Ein weiterer Hemmschuh dieser Art der Überdeckung liegt in der Beschränkung auf einzelne *GT*-Regeln. Der auf die Arbeiten von Stürmer et al. Bezug nehmende Ansatz [BKS04] von Baldan et al. ist dagegen primär als Ansatz zur Testgenerierung zu sehen. Die eingesetzte Abdeckung einzelner Regeln oder von Regelabhängigkeiten ist für das hier angestrebte Testen programmierter *GTs* ebenfalls ungeeignet.

Der Testansatz von Küster et al. aus [KA07] baut, wie bereits dargelegt, ebenfalls auf *GT*-Regeln auf. Die zugehörigen Muster umfassen abstrakte Elemente, welche durch kombinatorisches Einsetzen konkreter Typen aus dem Metamodell zu Testmodellen erweitert werden können. Dabei wird vornehmlich auf eine Überdeckung des Metamodells abgezielt. Bezogen auf die dabei genutzte Systematik weist der Ansatz aber gewisse Ähnlichkeiten zu einzelnen Strategien zur Konstruktion von Coverage-Items des im nächsten Kapitel vorgestellten Überdeckungsansatzes auf. Die ansonsten von Küster et al. betrachtete Abdeckung von Nebenbedingungen sowie die Untersuchung von Regelinteraktionen bzw. Überlappungen zur Ableitung von Tests haben für das hier verfolgte Ziel keine un-

mittelbare Relevanz. Gesonderte Nebenbedingungen werden hier nicht betrachtet und einzelne Regeln sind nicht frei kombinierbar.

Die Arbeit [DPV08] von Darabos et al. stellt einen sehr interessanten Ansatz zum Testen der unmittelbar ausführbaren, Code-basierten *Implementierung* einzelner *GT*-Regeln dar. Die Autoren betonen die Wichtigkeit einer Fokussierung auf die Korrektheit der Mustersuche und konzentrieren sich auf das Testen dieser, was sich direkt auf das hier betrachtete Szenario übertragen lässt. Das von Darabos et al. vorgestellte Fehlermodell zur Klassifizierung von Fehlern in der Implementierung der Mustersuche ist ebenfalls sehr relevant.

Den Kern des Ansatzes bildet die automatisierte Generierung von Testmodellen für einzelne *GT*-Regeln, die durch eine Interpretation der *LHS* der (Spezifikations-)Regel als Instanzgraph umgesetzt wird. Um dabei mehr als ein Testmodell zu erhalten, werden entsprechende Instanzgraphen auch anhand leicht modifizierter Variante der *LHSs* abgeleitet. Letztere entstehen durch Mutationen der ursprünglichen *LHS* auf Basis des Fehlermodells, wobei der Mutationsprozess durch einen Testansatz für Boolesche Ausdrücke, der ursprünglich dem Hardwaretesten entstammt, gesteuert wird.

Die Idee zur Nutzung von Variationen der *LHS* einer Regel für das Testen sowie verschiedene Aspekte der dazu genutzten Systematik werden uns auch nächsten Kapitel auch begegnen, allerdings in einem anderen Anwendungsfall. Denn im Gegensatz zu der Arbeit von Darabos et al. sollen keine Testmodelle generiert, sondern ein Überdeckungskonzept entwickelt werden. Auch ist die bei der Arbeit ebenfalls anzutreffende Beschränkung auf die Betrachtung einzelner isolierter Regeln nicht zielführend. Darüber hinaus sind auch die Interpretationen von *GT*-Regeln und des Testproblem leicht anders: bei Darabos et al. ist eine Regel als (Test-)Spezifikation zu verstehen und eine in einer klassischen Programmiersprache umgesetzte Implementierung ist das *SUT*. Hier ist dagegen die aus den Regeln zusammengesetzte Beschreibung bereits die Implementierung und damit als *SUT* anzusehen und der Codegenerator oder ein hypothetischer Interpreter sind als fehlerfrei anzunehmen. Nichtsdestotrotz beschreiben Darabos et al. wichtige Ideen und Konzepte, die sich sowohl für das Überdeckungskonzept für programmierte *GTs* als auch für den *SDM*-spezifischen Mutationsansatz aufgreifen lassen.

Eine Schwäche von [DPV08] liegt darin, dass wichtige Teilaspekte, wie die Erzeugung der mutierten Varianten mittels sog. „Metatransformation Rules“ nur skizziert werden, und dass wesentliche praktische Herausforderungen, z. B. was wird wie genau Mutiert, wie können inkonsistente Modelle vervollständigt oder ggf. proaktiv verhindert werden, außen vor bleiben. Auch bleibt eine Betrachtung der Güte der so ableitbaren Tests sowie des praktischen Nutzens des Verfahrens außen vor. Eine Beschreibung einer prototypischen Implementierung (mit starkem VIATRA2-Bezug) sowie erste Ergebnisse einer praktischen Erprobung, welche zumindest die grundsätzliche Anwendbarkeit exemplarisch belegt (Aspekte der Leistungsfähigkeit der Tests bleiben offen), lassen sich der Masterarbeit von Darabos [Dar07] entnehmen.

Die einzige mir zu diesem Zeitpunkt bekannte existierende Arbeit mit unmittelbarem Bezug zum Testen von programmierten *GTs* im Allgemeinen und dem Testen von Story-Diagramm-basierten Transformationen im Speziellen stammt von Geiger, vgl. dazu seine Dissertation [Gei11]. Relevanz für das hier verfolgte Ziel besitzt der von Geiger beschriebene Szenario-basierte Testansatz, wenn auch dieser vornehmlich für den Einsatz in Entwicklungsprojekten nach dem proprietären sog. FUP (FUjaba Process) erdacht wurde. Der Ansatz basiert auf konzeptuellen, FUP-spezifischen Szenario-Beschreibungen,

den sog. *Storyboards*. Diese nutzen GT-Regeln zur skizzenhaften Beschreibung von Vor- und Nachbedingungen einer (Teil-)Funktionalität, können aber auch schrittweise per Codegenerierung mittels Fujaba-Erweiterung, über den Zwischenschritt einer auf Story-Diagrammen aufbauenden Operationalisierung, in JUnit-Tests übersetzt werden. Vordergründig handelt es sich also um einen funktionsbasierten Black-Box-Testansatz. Allerdings werden in diesem Zusammenhang von Geiger auch die auf/ab S. 135 bereits erwähnten Formen der Abdeckung skizziert (die wahrscheinlich unabhängig von den Storyboards zu sehen sind). Mit ihnen soll untersucht werden, ob die Implementierung Teile umfasst, die noch nicht mittels Szenario-basierten JUnit-Tests getestet wurden. Naheliegender wäre es allerdings, die Abdeckung zu nutzen, um – *unabhängig von den Szenarien* – abzuschätzen, wie gründlich eine Story-Diagramm-basierte Implementierung getestet wurde.

Insbesondere Geigers Ausführungen, dass bei einer Abdeckung des Kontrollflussgraphen die beiden möglichen Auswertungsergebnisse der Regelknoten zu berücksichtigen seien sowie einige der Aussagen bei der Beschreibung der verschiedenen Stufen seiner *Search Coverage* lassen sich auch auf das hier verfolgte Ziel übertragen. So repräsentieren die Aussagen, dass einzelne Regeln nicht als atomare Einheit zu sehen seien [Gei11, S. 73], dass „die Graphsuche zumindest einmal inklusive aller optionalen [...] Knoten erfolgreich sein muss“ [Gei11, S. 74] oder dass die beteiligten Metamodelle Einklang in das Überdeckungskonzept finden sollten [Gei11, S. 75] wichtige Ideen. Diese spiegeln sich auch zum Teil in dem im nächsten Kapitel vorgestellte Überdeckungskonzept wieder. Dagegen erscheint der von Geiger verfolgte Weg der Umsetzung, zumindest dem hier vertretenen Standpunkt nach, suboptimal. Statt zu versuchen, die einfache Anweisungsüberdeckung auf Codeebene in eine Überdeckung von GT-Regelteilen bzw. Suchoperationen zurückzuübersetzen, wäre ein Ansatz ohne Bezug zu einer codebasierten Überdeckung vorzuziehen. Auch wird nicht deutlich genug, inwiefern das Erkennen einer zu geringen Überdeckung auf Codeebene zu sinnvollen Erweiterungen auf Seite der Tests bzw. Testszenarien führt – Kenntnis der Codestruktur erscheint z. B. wichtig zu sein. Geiger selbst merkt dazu an, dass die von ihm vorgestellten Überdeckungskriterien hinsichtlich ihrer Tauglichkeit untersucht werden könnten, vgl. [Gei11, S. 79]. Ein Abschnitt zu den Erfahrungen aus dem Einsatz im Rahmen mehrerer studentischer Projekte spart diesen Aspekt aus.

6.4.2 Herausforderungen und offene Punkte

Zusammenfassend bleibt festzuhalten, dass zwar eine stetig wachsende Menge an ausgereiften, funktionsbasierten Testansätzen für MT-Programme entwickelt wird, die grundsätzlich wiederverwendbar sind, dazu aber die Struktur und Eigenheiten der Implementierung bewusst ausklammern. White-Box-Überdeckungskonzepte sind dagegen vor allem für die populäre (textuell) MT-Sprache ATL entwickelt worden. Überdeckungskonzepte für *MTs* auf Basis von Kontrollflussgraphen greifen für den hier zu betrachtetenden Anwendungsfall zu kurz, da die entscheidenden GT-Aspekte der Implementierung unberücksichtigt bleiben würden. Klassische codebasierte Überdeckungsansätze erscheinen schon aufgrund ihres Abstraktionsniveaus als unpassend und sind weitestgehend unbrauchbar bei einer Interpreter-basierten Ausführung.

Ansätze speziell zum Testen von *GTs* sind rar, wobei sich die überwiegende Mehrzahl derer auch noch auf die Betrachtung isolierter GT-Regeln oder von Mengen in beliebiger Reihenfolge ausführbarer Regeln beschränkt. Die beiden im letzten Abschnitt ausführ-

licher betrachteten Arbeiten mit der größten inhaltlichen Nähe zum hier verfolgten Ziel beschreiten beide interessante Wege. Sie weisen allerdings auch nachteilige Eigenschaften auf, die einer direkten bzw. vollständigen Wiederverwendung entgegenstehen. Entscheidend ist, dass ein neues Überdeckungskonzept für programmierte *GTs* im Allgemeinen und *SDM*-Transformationen im Speziellen gewisse Eigenschaften aufweisen sollten. Es sollte praktikabel sowie flexibel erweiterbar sein, die innere Struktur der Muster und die Mustersuche gebührend berücksichtigen, dabei aber nicht den Kontrollflusskontext ignorieren und sowohl für Code-basierte als auch Interpreter-basierte Realisierungen geeignet sein. Darüber hinaus sind im Idealfall Aspekte der praktischen Anwendbarkeit und insb. eine realistische Abschätzungen zum Zusammenhang zwischen Leistungsfähigkeit und erreichtem Abdeckungsgrad (möglichst adäquater Testmengen) bereits geklärt und als günstig bewertet worden. Diese Eigenschaften bilden nun die Ausgangslage für die folgende Betrachtung der Lösungsvorschläge und Beiträge.

Teil II

Beiträge

> Bug prevention is testing's first goal. [...] the act of *designing* tests is one of the best bug preventers known.
>
> *(B. Beizer, aus [Bei90])*

7 Testen von SDM-Transformationen

Dieses Kapitel ist der Vorstellung von einem der Hauptbeiträge der Arbeit gewidmet. Inhaltlich geht es dabei um die Herausforderung, vollständige Transformationen in Form programmierter Graphtransformationen zu testen. Insbesondere soll hierdurch die Frage beantworten werden, wann genug getestet wurde und welche wesentlichen Testfälle ggf. noch fehlen und ggf. ergänzt werden sollten. Dazu wird ein konkreter Ansatz auf Basis eines neuen Überdeckungskriteriums für programmierte *GTs* am Beispiel der *SDM*-Sprache vorgestellt. Dieses Kapitel greift Konzepte aus der Arbeit [WS13] auf und erweitert und ergänzt diese zu einer umfangreichen und verallgemeinerten Darstellung.

Der Kapitel ist wie folgt aufgebaut: In Abschnitt 7.1 werden die initialen Herausforderungen bezogen, auf die Entwicklung des gesuchten Überdeckungskriteriums, ausgearbeitet und entsprechende Anforderungen an ein Überdeckungskriterium für programmierte *GTs* entwickelt und beschrieben. Im Anschluss daran wird in Kapitel 7.2 der von mir entwickelte Ansatz zuerst übersichtsartig skizziert und im Anschluss detailliert ausgearbeitet. Der Fokus liegt dabei auf der allgemein gehaltenen Vorstellung der Konstruktionsvorschrift zur Erzeugung der benötigten Coverage-Items, welche in Form eines Algorithmus präsentiert wird, sowie auf der detaillierten Beschreibung und Motivation der den Algorithmus konkretisierenden Strategien zur Erzeugung der konkreten Coverage-Items. Anschließend werden in Unterkapitel 7.3 Aspekte einer konkreten Implementierung, wie eine Unterstützung durch ein entsprechendes Werkzeug, betrachtet sowie auf wichtige technische Details eingegangen. Abschnitt 7.4 ist der Anwendung des umgesetzten Ansatzes im Kontext der bereits vorgestellten Beispieltransformation gewidmet. Abgeschlossen wird das Kapitel von einer kurzen zusammenfassenden Betrachtung und Bewertung des bis dato in Abschnitt 7.5 Vorgestellten.

7.1 Herausforderungen und resultierende Anforderungen

Ein wesentlicher Teil der Komplexität bei der Entwicklung mit einer deklarativen Graphtransformationssprache liegt in der korrekten Formulierung des Musteranteils, also der Vorbedingung, einer Regel, vgl. [DPV08]. In der Praxis entstehen viele Fehler dadurch, dass entweder unerwünschte oder gar keine Elemente des Modells für die jeweiligen Variablen des Musters identifiziert werden. Dies kann dann in ausbleibenden, unnötigen oder falschen Änderungen im Modell resultieren, vgl. ebenfalls [DPV08].

In Abbildung 7.1 sind vier mögliche Arten von Beziehungen zwischen einem als korrekt anzusehenden, idealen Muster P und einer tatsächlich vorliegenden Realisierung P' als Mengendiagramme qualitativ skizziert, vgl. dazu z. B. auch [Wim+09, Abb. 5]. Dabei sind die folgenden Fälle grundsätzlich zu unterscheiden, in denen das Muster P' (a) zu *restriktiv* (eigentliche Treffer werden übersehen), (b) zu *liberal* (es werden vermeintliche „Treffer“ erkannt, deren Struktur oder Eigenschaften fast richtig sind), (c) in Teilen fehlerhaft (einige der vermeintlichen Treffer weisen erheblich Unterschiede zu den eigentlich erwünschten Treffern auf) oder (d) grundlegend falsch ist (die Mengen der tatsächlichen und der eigentlich benötigten Treffer sind disjunkt).

Im Falle eines zu restriktiven Musters, vgl. Abbildung 7.1a, werden eigentlich passende Modellstrukturen, dargestellt durch den grau eingefärbten Bereich der Menge P, als nicht geeignet klassifiziert und aussortiert. Allerdings handelt es sich bei allen Treffern für P' tatsächlich um valide Treffer des eigentlich korrekten Musters.

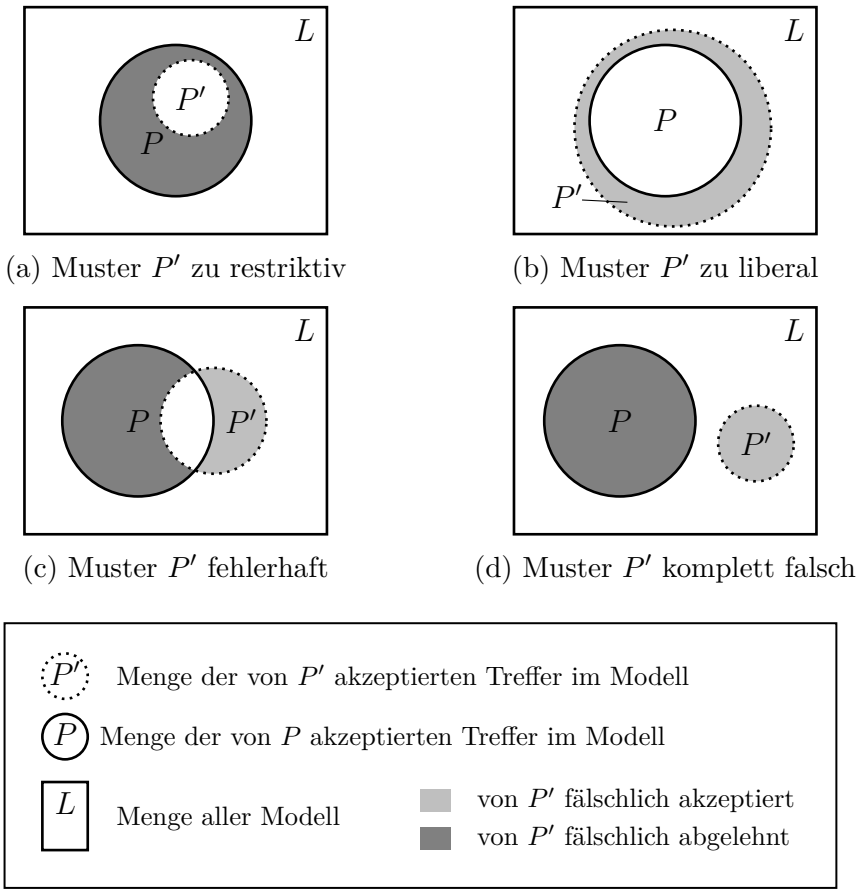
Zu liberale Muster, vgl. Abbildung 7.1b, führen dagegen dazu, dass die vorgegebene Struktur im Modell auch zu eigentlich unerwünschten Treffern führen kann. Tendenziell werden also zu viele und auch falsche Treffer gefunden. Teilweise fehlerhafte, wie Teilabbildung 7.1c, oder komplett falsche Muster, wie in Teilabbildung 7.1d, sind mehr oder weniger offensichtlich fehlerhaft, da sie das eigentlich angestrebte Verhalten seltener oder gar nicht aufweisen können.

Dass Fehler der Art (c) oder (d) erkannt werden, ist i. d. R. wahrscheinlicher als dass dies bei den anderen beiden Fehlerarten der Fall ist, da dort die Unterschiede subtiler sind und tendenziell nur für Randfälle auftreten. Das bedeutet aber auch, dass sich das Testen insbesondere auf diese letzteren, subtileren Fälle eingehen sollte, da diese ansonsten leicht übersehen werden können. Aus diesen Erkenntnissen lässt sich eine erste Anforderung an einen Testansatz ableiten:

Anforderung 7.1

Graphmuster (und -regeln) sind komplexe Gebilde mit einer eigenen Struktur und hierdurch bedingter komplexer (Auswertungs-)Semantik. Die Struktur eines Musters sollte folglich beim Testen ausreichend berücksichtigt werden.

Wie in Abschnitt 4.3 dargelegt, wird bei *SDM*-Transformationen die Transformation durch eine Menge von *GT*-Regeln beschrieben, die über den Kontrollflussgraphen miteinander verknüpft sind, welcher ihre Auswertungsreihenfolge und Abhängigkeiten beschreibt. Ob sich aufgrund einer Regelanwendung ein Fehlerfall einstellt, und wenn ja, welcher Art ein solcher Fehler ist, lässt sich also nicht ausschließlich anhand der isolierten Regel entscheiden, da stets auch der Kontrollflusskontext betrachtet werden muss. Somit erscheint ein solcher *struktureller Testansatz* sinnvoll, der sich auf die Details und die Struktur der gesamten Implementierung stützt, also auf die Graphtransformationsregeln



(a) Muster P' zu restriktiv

(b) Muster P' zu liberal

(c) Muster P' fehlerhaft

(d) Muster P' komplett falsch

Legende

Abbildung 7.1: Beziehungen zwischen korrektem (P) und realisiertem (P') Muster

unter Berücksichtigung des Kontrollflusses. Hieraus ergibt sich eine weitere Anforderung an einen Testansatz:

Anforderung 7.2

Ein Überdeckungskriterium *für programmierte Graphtransformationen (am Beispiel von SDMs) sollte* struktureller Art *sein und die wesentlichen Details der Implementierung berücksichtigen.*

Für die Entwicklung eines Überdeckungsansatzes für *SDM*-Transformationen stellt sich die Frage nach der Granularität des Testens. Ein praktikabler Ansatz sollte sich auf geeignet große Teile des zu testenden Systems konzentrieren. Isolierte Graphmuster sind, wie im vorangegangenen Abschnitt ausgeführt, für den hier betrachteten Anwendungsfall ungeeignet, da die Mustersuche immer im Kontext der aktuellen Position im Kontrollflussgraphen erfolgt. Als weitere Einschränkung ergibt sich Anforderung 7.3.

Anforderung 7.3

Die Auswertung der zu definierenden Überdeckung muss einzelne Graphregeln, insbesondere die zugehörigen Musteranteil der LHSs, in ihren jeweiligen Kontrollflusskontexten *betrachten, in denen sie ausgewertet werden.*

Darüber hinaus stellt eine Operation, deren Verhalten durch ein einzelnes *SDM*-Diagramm festgelegt wird, eine in sich geschlossene, funktionale Einheit der Transformationsimplementierung dar. Folglich liegt es nahe, Testanforderungen mit unmittelbarem Bezug zu den einzelnen Operationen zu formulieren und zu verwalten. Als Anforderung formuliert, ergibt sich die nachfolgende Anforderung 7.4.

Anforderung 7.4

Der Testansatz und das zu entwickelnde Überdeckungskriterium sollte einen Bezug zu den Operationen im Metamodell herstellen, so dass sich Testfälle und Testanforderungen auf einzelne Operationen beziehen können.

Das Überdeckungskriterium soll sich allerdings *nicht* ausschließlich auf den Kontrollfluss beziehen, da ansonsten die zentralen Aspekte der Graphtransformationen unberücksichtigt bleiben würden. Auch lassen sich etablierte Überdeckungskriterien für Kontrollflussgraphen i. Allg., vgl. Kapitel 5.3.2, oder ggf. Aktivitätsdiagramme im Speziellen, vgl. Kapitel 6.2.2, übertragen. Datenflussbasierte Ansätze sind grundsätzlich denkbar, erscheinen aber aufgrund des impliziten Bindens von Variablen beim Pattern-Matching als schwerlich übertragbar. Zusammenfassend ergibt sich Anforderung 7.5.

Anforderung 7.5

Das Überdeckungskriterium soll nicht *über den Kontroll- oder den Datenfluss definiert sein, da ansonsten Komplexität und entsprechende Fehlerquellen in den GT-Regeln unberücksichtigt bleiben würden.*

Allerdings sollte durch das verwendete Überdeckungskriterium sichergestellt werden, dass jedes Muster durch mindestens einen Test auch zur Anwendung kommt. Sinnvoll erscheint zusätzlich die Forderung, dass jedes Muster im Verlauf des Testens *mindestens einmal mit positivem* und *einmal mit negativem Resultat* ausgewertet werden muss, vgl. hierzu auch [Gei11]. Interpretiert man ein Muster als Verzweigungspunkt eines Kontrollflussgraphen, so sollte jeder Verzweigungspunkt erreicht und mit jedem der beiden möglichen Ergebnissen ausgewertet werden. Dieser Zusammenhang begründet Anforderung 7.6.

Anforderung 7.6

Ein Überdeckungskriterium sollte möglichst auch Knotenüberdeckung *und gegebenenfalls auch* Kantenüberdeckung *subsumieren, jeweils bezogen auf den Kontrollflussteil der GT-Spezifikation.*

Metamodelle können viele verschiedene Klassen oder komplexe Vererbungs- und Assoziationsbeziehungen umfassen. Dies kann zu einer großen Variabilität auf Instanz- bzw. Modellebene führen, was unmittelbar eine großen Vielfalt von Mustern selbst bei keinen Regeln führt. Auch können konkrete Treffer im Modell ganz unterschiedliche Ausprägungen aufweisen. Beispielsweise können konkrete Objekte vom Typ einer beliebigen Unterklasse sein oder *OVs* können durch mehrere potentiell mögliche Assoziationen miteinander verknüpft sein. Ein Überdeckungskonzept sollte der metamodellbedingten Variabilität Rechnung tragen, indem möglichst verschiedenartige Testmodelle eingefordert werden, vgl. hierzu auch die Ideen der Partitionenüberdeckung in Abschnitt 5.3.4 oder der Metamodell-Überdeckung in Abschnitt 6.2.1. Kompakt zusammengefasst, ergibt sich daraus Anforderung 7.7.

Anforderung 7.7

Das zu entwickelnde Überdeckungskonzept sollte sicherstellen, dass möglichst verschiedenartige Modelle zum Testen verwendet werden, sodass möglichst die volle Variabilität aufgrund des Metamodells abgedeckt wird.

Die letzte Anforderung 7.8 betrifft einen technischen Aspekt bei der Umsetzung und ergibt sich aus der bereits dargelegten Forderung nach der Unabhängigkeit von jedweder Codegenerierung zur Ausführung der *GT*.

Anforderung 7.8

Der Überdeckungsansatz soll unabhängig davon sein, ob aus der Transformation Code generiert wird oder ein Interpreter zur Ausführung genutzt wird. Die Definition der Coverage-Items sollte auf einer Ebene mit der GT-Beschreibung liegen und nicht über ein „darunterliegendes“ Zwischenformat, wie der hier vorkommende Java-Code.

7.2 Der Überdeckungsansatz

Auf Grundlage der zuvor identifizierten Anforderungen wird im Folgenden das Überdeckungskonzept entwickelt. Hierzu wird in Abschnitt 7.2.1 zuerst die grundlegende Idee des

Ansatzes skizziert. Danach wird in Abschnitt 7.2.2 das entsprechende Überdeckungskriterium definiert, bevor dann im Abschnitt 7.2.3 die Details der Konstruktionsvorschriften zur automatisierten Erzeugung der benötigten Coverage-Items beschrieben werden. Hier bleibt noch festzuhalten, dass der Ansatz, je nach Blickwinkel, gewisse Ähnlichkeiten mit dem einen oder anderen klassischen Überdeckungskonzept aufweist. Beispielsweise werden, vereinfacht ausgedrückt, aus den Graphmustern weitere Vorbedingungen abgeleitet und in Form von parametrisierten Aussagen bzw. Prädikaten, vgl. Definition 5.19, zur Definition der Testabdeckung genutzt. Dies ist in gewisser Weise ähnlich zum Vorgehen bei den Ansätzen zur logikbasierten Überdeckung aus Kapitel 5.3.3. Die Methodik, mit der die Vorbedingungen tatsächlich abgeleitet werden, zeigt dagegen Parallelen mit Verfahren des klassischen Mutationstestens. Darüber hinaus wurde der Aspekt der Subsumption, beispielsweise von Kontrollflussüberdeckungskriterien, bereits erwähnt. Aufgrund des Kontrollflussteils der *SDM*-Sprache sind entsprechende Relationen ebenfalls möglich und grundsätzlich untersuchenswert. Diese erwähnten Parallelen wurden bei der Entwicklung nicht aktiv verfolgt bzw. angestrebt. Ihr Vorhandensein deuten aber darauf hin, dass der hier vorgestellte Überdeckungsbegriff brauchbar sein könnte. Dies zu Belegen ist wesentlicher Inhalt der nachfolgenden Kapitel.

7.2.1 Eine musterbasierte Testüberdeckung

Nachdem bereits dargelegt wurde, dass die Mustersuche wesentlich ist für die Korrektheit der gesamten *SDM*-Transformation, werden wir hier einen auf Graphmustern basierenden Testansatz vorstellen, der dieser Tatsache Rechnung trägt. Dem Ansatz liegt eine einfache Erkenntnis zu Grunde: die *LHS* einer Graphersetzungsregel stellt einerseits die Vorbedingung zur eigentlichen Regelanwendung und andererseits den wesentlichen Kernteil der Entscheidung über eine eventuelle Kontrollflussverzweigung dar. Durch ein Überdeckungskriterium sollte also sichergestellt werden, dass jede Graphersetzungsregel im Rahmen des Testens mindestens einmal zur Ausführung kommt und angewendet wird. Diese Forderung lässt sich so umformulieren, dass verlangt wird, dass für die *LHS* der Regel mindestens einmal ein Match im Objektgraphen gefunden wird. Außerdem sollte das Überdeckungskriterium sicherstellen, dass für jede Graphersetzungsregel nach Möglichkeit auch mindestens einmal der Fall vorkommt, bei welchem die Vorbedingung für eine Anwendung nicht erfüllt ist. Dies entspricht der Forderung, dass für den Musteranteil jeder Regel mindestens einmal *kein* Match im Objektgraphen gefunden werden soll.

Möchte man allerdings wirklich nachvollziehen, ob die Graphmuster der vorliegenden Implementierung tatsächlich das zu lösende Problem hinreichend genau beschreiben und entsprechend lösen, muss man sie mit *ausreichend vielen* und *guten* Testmodellen testen. Gut bedeutet, dass von den vielen potentiell möglichen Modellen, auf denen das Muster angewendet werden kann, eine endliche Auswahl an repräsentativen Fällen so gewählt wird, dass ein Entwickler *begründetes Vertrauen* in die Korrektheit seiner Lösung haben kann. Dies ist dann der Fall, wenn sich das Muster beispielsweise in Situationen bewährt hat, in denen es „fast passende“ Teilstrukturen im Objektgraphen korrekterweise ignoriert hat oder in denen es möglichst verschiedenartige legale Varianten passender Teilstrukturen sicher erkannt hat. Aufgrund einer der Kernhypothesen des Mutationstestens, nämlich dass ein Entwickler *kompetent* ist, er also dazu tendiert, Lösungen zu entwickeln, die, falls falsch, so zumindest relativ nahe an einer richtigen Lösung gelegen,

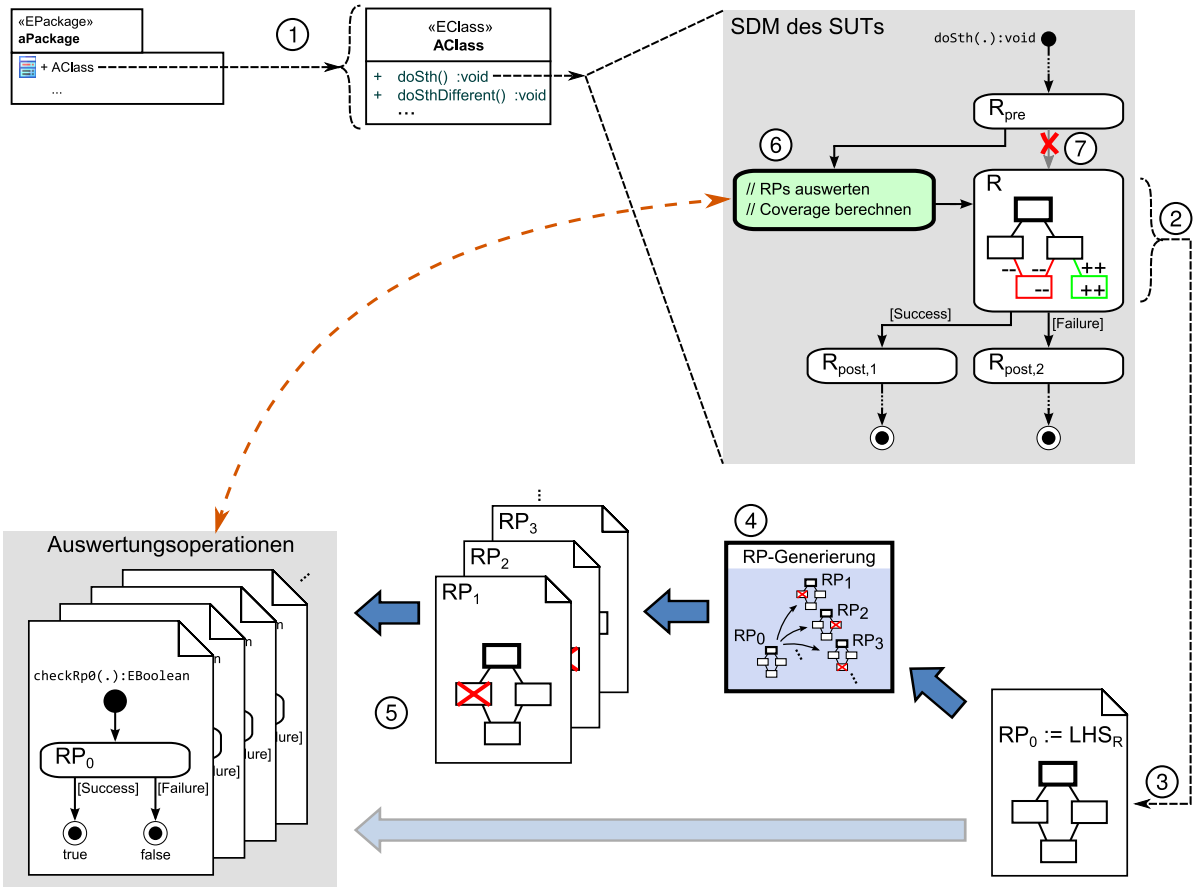


Abbildung 7.2: Die wichtigsten Schritte beim *RP*-basierten Testen

s. [DLS78], kann man auch annehmen, dass spezifizierte Muster mehr oder weniger gute Näherungen für die korrekten Varianten sind. Ein fehlerhaftes Muster unterscheidet sich folglich vielleicht nur sehr geringfügig von einem Korrekten.

Um den Fokus auf solche potentiell feinen Unterschiede zwischen „korrekt“ und „nicht mehr ganz korrekt“ zu lenken und um darüber hinaus das Vorhandensein möglichst relevanter Objektstrukturen im Modell vor der Auswertung der betrachteten Regel einzufordern, werden hier aus den *LHSs* der Graphtransformationsregeln Hilfsmuster, die sog. *Requirement-Patterns (RPs)*, abgeleitet. Letztere entstehen jeweils durch systematisches Ableiten aus der *LHS* einer der Regeln und repräsentieren die Bausteine zur Formulierung der benötigten Coverage-Items.

In Abbildung 7.2 ist eine Übersicht über diesen Ableitungsvorgang dargestellt. Dieser umfasst im Wesentlichen 7 Teilschritte. In Schritt ① wird das Metamodell traversiert und es werden alle Klassen, die Operationen definieren, deren Verhalten mit Hilfe einer *SDM*-Beschreibung spezifiziert ist, gesammelt. Die einzelnen *SDM*-Implementierungen werden untersucht und dabei die einzelnen *GT*-Regeln weiter verarbeitet. In Schritt ② wird über die einzelnen Regeln iteriert – im Beispiel durch eine bestimmte Regel R repräsentiert – und aus diesen das jeweilige Muster in Schritt ③ extrahiert. Die *LHSs* selbst können bereits als einzelne *RPs* aufgefasst werden; hier und im Folgenden werden diese als RP_0 bezeichnet. Sie dienen einer Komponente zur systematischen Ableitung weiterer *RPs* als Eingabe, vgl. Punkt ④. In diesem Beispiel ist angedeutet, dass die *RPs* durch das Um-

wandeln einzelner obligatorisch verlangter Knoten in *NAC*-Knoten erzeugt werden. Dies ist nur eine von vielen Möglichkeiten und ist hier nur exemplarisch zu verstehen. Mehrere solcher Konstruktionsvorschriften werden im übernächsten Unterabschnitt vorgestellt.

Sobald eine Menge an *RPs* vorliegt, werden diese in Teilschritt ⑤ jeweils in *SDM*-Schablonen eingefügt, wodurch die Implementierung einer Auswertungsoperation für das jeweilige *RP* entsteht. Aufgrund des Booleschen Rückgabewertes der Operationen können diese als Prädikate über den Variablen des Musters aufgefasst werden, die für einen bestimmten Zustand – den Modellgraphen bei ihrem Aufruf – ausgewertet werden. Der Kontext der Auswertung ergibt sich anhand der Wertebelegung der Parameter der Operation. In der Abbildung soll entsprechend durch den Punkt in der Signatur der Operation angedeutet werden, dass die Liste der Parameter i. Allg. nicht leer ist. Die benötigten Parameter, welche die entsprechende *OVs* an Objekte des Modells binden, ergeben sich aus dem konkreten Aufbau des jeweiligen Musters. Warum aus den *RPs* gerade *SDM*-basierte Operationen generiert werden, sollte spätestens im nächsten Abschnitt deutlich werden. Neben technischen Gründen spricht auch die Kombinierbarkeit der checkRpX(.) Auswertungsoperationen hierfür.

Damit die Überdeckung zur Laufzeit bestimmt werden kann, müssen die abgeleiteten Auswertungsoperationen an den richtigen Stellen der ursprünglichen *SDM*-Implementierung ausgewertet werden. Hierzu muss die ursprüngliche Transformationsbeschreibung entsprechend angepasst und durch Instrumentierungsanweisungen ergänzt werden. In Schritt ⑥ wird dazu ein neuer Knoten in der ursprünglichen *SDM* ergänzt, der anschließend dafür sorgt, dass die abgeleiteten Auswertungsoperationen aufgerufen *und* entsprechende Überdeckungsstatistiken im Anschluss an die Auswertung aktualisiert werden. Als letzter Schritt muss noch der ursprüngliche Kontrollfluss so abgepasst werden, dass die *RPs* unmittelbar vor der eigentlichen Regel ausgewertet werden, vgl. Schritt ⑦. Da *RPs* per Definition nur Graphmuster darstellen, folglich auch keine Ersetzungschritte definieren und damit frei von Seiteneffekten sind, verändern sie das funktionale Verhalten der Transformation nicht. Allerdings muss mit einer Verschlechterung des Laufzeitverhaltens der Transformation gerechnet werden.

Auf Basis der abgeleiteten *RPs* und den sich daraus ergebenden Auswertungsoperationen lässt sich ein Überdeckungsmaß dergestalt definieren, dass man verlangt, dass jedes Prädikat mindestens einmal zu `wahr` und einmal zu `falsch` evaluiert. Entsprechende Definitionen folgen im nächsten Abschnitt 7.2.2. Darüber hinaus könnten, wie bereits angedeutet, Prädikate komponiert werden, um weitere Testanforderungen zu bilden. Hierbei ist allerdings zu beachten, dass die Auswahl der Modellelemente, an die die Variablen von Mustern gebunden werden, nichtdeterministisch erfolgt. So erfolgt das Binden bei der Auswertung von zwei oder mehreren Mustern unabhängig voneinander. Das hat zur Konsequenz, dass man nicht einfach überprüfen kann, ob ein einmal identifizierter Treffer eines bestimmten *RP* auf dem gegeben Modellgraphen auch einen Match für ein anderes *RP* darstellt. Ist eine solche Überprüfung nun verlangt, z. B. im Rahmen der Komposition von *RPs*, so müssten die Teilmatches für solche Variablen, die sich die betrachteten *RPs* teilen, zwischen den Auswertungen kommuniziert und ausgetauscht werden. Dies ist in der aktuellen Implementierung der Mustersuche so nicht vorgesehen. Nichtsdestotrotz lassen sich auch ohne diese Möglichkeit bestimmte *RPs* auf sinnvoll Art kombinieren.

Ist das abgeleitete *RP* beispielsweise durch eine Auflockerung der Vorbedingungen des originären Musters entstanden, vgl. hierzu Abbildung 7.1b, so sind vor allem Testmo-

delle interessant, auf denen das originäre Muster nicht matchen würde, das abgeleitete *RP* aber schon. Diese Modelle entstammen aus dem mit Grau hinterlegten Teil aus Abbildung 7.1b, wenn man das abgeleitete *RP* als P' und das originäre Muster als P interpretiert. Nur bei Eingaben aus diesem Bereich sollte es zu unterschiedlichen Entscheidungen bei der Suche nach den beiden jeweiligen Mustern kommen. Somit sind Eingaben dieser Art grundsätzlich als Grenzfälle für die betrachtete Regel anzusehen und äußerst relevant für das Testen. Allerdings sei an dieser Stelle auch betont, dass es hierbei nicht um Ergebnisse der Auswertung geht, also auch keine Ableitung von Nachbedingungen oder anderen Zusicherungen erfolgt, die für eine *Bewertung* der Testerausgaben herangezogen werden könnten. Eine Überprüfung, ob es sich um einen Fehler oder gewünschtes Verhalten handelt, wenn sich *RP* und originäres Muster bzgl. der Trefferfindung unterschiedlich verhalten, kann im allgemeinen nicht automatisiert erfolgen. Auch die Frage, ob es überhaupt möglich ist, ein Eingabemodell so zu erstellen, dass es zum entsprechenden Punkt im Ablauf der *SDM* die benötigten Eigenschaften aufweist, ist nicht abschließend zu beantworten. Um die Existenz eines Testmodells aus dem Bereich $P' \setminus P$ durch ein Coverage-Item sicherzustellen, muss eine entsprechende Bedingung formuliert werden, die dafür sorgt, dass für RP_0, welches ja dem originären Muster (in der Abbildung P) entspricht, kein Match gefunden werden kann, was der Teilbedingung `checkRp0(.) == false` entspricht, und dass für RP_i, das hier das abgeleitete *RP* (P' in der Abbildung) bezeichnet, ein entsprechender Match gefunden werden kann, was der Teilbedingung `checkRpi(.) == true` entspricht. Folglich würde die komplette Bedingung eines solchen Coverage-Items lauten: `(!checkRp0(.) && checkRpi(.)) == true`.

Ist das abgeleitete *RP* durch eine Verschärfung der Vorbedingungen des originären Musters entstanden, sind Testmodelle relevant, die dazu führen würden, dass das *RP* nicht gefunden wird, das originäre Muster aber schon. Die Bedingung für ein entsprechendes Coverage-Item lautet folglich `(checkRp0(.) && !checkRpi(.)) == true`.

Wurde das *RP* durch eine Ableitungsvorschrift gebildet, die zu einer der Situationen wie in den Abbildungen 7.1c oder 7.1d führt, so sind im besonderen Maße solche Testmodelle von Interesse, die für eine der beiden Varianten des Musters in einer Übereinstimmungen resultieren, für die andere Variante aber nicht, und umgekehrt. Zusammengefasst bedeutet dies, dass `((checkRp0(.) && !checkRpi(.)) == true)` für eine Auswertung der *RPs* gelten sollte sowie für eine weitere Auswertung umgekehrt `((!checkRp0(.) && checkRpi(.)) == true)`.

7.2.2 Das RP-Überdeckungskriterium

Auf der Grundlage der abgeleiteten *RPs*, kann nun ein Überdeckungsmaß, vgl. Definition 5.11, für programmierte Graphtransformationen am Beispiel von *SDMs* festgelegt werden. Dazu wird die Menge der Anforderungen *TR* vorläufig so festgelegt, dass sie genau eine Anforderung für jedes *RP* aus der relevanten Teilmenge aller generierten *RPs* für das gesamte *SUT* enthält. Dabei ergibt sich die konkrete Teilmenge der generierten *RPs* anhand der kleinsten Einheit des Testens, also einer einzelnen Regel. In der nachfolgenden Definitionen 7.1 bis 7.3 werden die entsprechenden Überdeckungskriterien auf Basis der zuvor beschriebenen *einfachen* Coverage-Items festgelegt.

Definition 7.1 (Einfache *positive RP*-Überdeckung):

Als einfache positive *RP-Überdeckung c^+ einer Graphersetzungsregel wird das Verhältnis aus Anzahl der aus der Regel abgeleiteten RPs, für die im Laufe des Testvorgangs mindestens einmal ein Match gefunden wird, zu der Anzahl der insgesamt für die Regel abgeleiteten RPs festgelegt.*

Neben der Forderung aus Definition 7.1, dass Prädikate mindestens einmal zu `wahr` evaluieren sollen, ergibt sich eine weitere naheliegende Forderungen: Die selben Prädikate sollen im Rahmen des Testens auch mindestens einmal zu `falsch` evaluieren. Dies ist in der sich anschließenden Definition 7.2 festgehalten.

Definition 7.2 (Einfache *negative RP*-Überdeckung):

Analog zu Definition 7.1, bezeichnen wir das Verhältnis aus der Anzahl der aus einer Regel abgeleiteten RPs, für die im Laufe des Testvorgangs mindestens einmal kein *Match gefunden werden kann und der Anzahl der insgesamt für die Regel abgeleiteten RPs als einfache* negative *RP-Überdeckung c^- einer Graphersetzungsregel.*

Beide Überdeckungsmaße lassen sich zu einer kombinierten Größe vereinen, wie in in Definition 7.3 festgelegt. Anschaulich wird durch dieses kombinierte Maß gefordert, dass für jedes *RP* mindestens ein Testfall existiert, der dafür sorgt, dass das *RP* im Rahmen seiner Auswertung(en) mindestens einmal im Modellgraphen gefunden wird, und dass mindestens ein Testfall existiert, bei dem für das *RP* im Rahmen seiner Auswertung kein Match im Modellgraphen gefunden werden kann. Dabei müssen diese beiden Testfälle nicht notwendigerweise verschieden sein.

Definition 7.3 (Einfache *RP*-Überdeckung):

*Als RP-*Überdeckung *oder Requirement Pattern Coverage (RPC) bezeichnen wir ein Maß c, das sich als Kombination der Maße aus den Definitionen 7.1 und 7.2 ergibt, und dessen Wert gleich dem des arithmetischen Mittels aus c^+ und c^- ist: $c = \frac{c^+ + c^-}{2}$.*

Gegen Ende des Abschnitts 7.2.1 wurde, ab Seite 152, auf die Komponierbarkeit von Prädikaten zur Definition weiterer Coverage-Items hingewiesen. Solche zusammengesetzten Prädikate sind bei den obigen Definitionen noch nicht erfasst. Die Definition eines entsprechenden Überdeckungsmaßes ist allerdings unmittelbar möglich.

Definition 7.4 (*RP*-Überdeckung (allg.)):

Die RP-Überdeckung $\tilde{c}$ einer GT-Regel ist das Verhältnis aus der Anzahl von überdeckten Coverage-Items, welche als einfache/zusammengesetzte Prädikate über den abgeleiteten RPs formuliert sind, zur Gesamtzahl der Coverage-Items.

Ein Coverage-Item für ein Prädikat, egal ob einfach oder zusammengesetzt, positiv oder negativ, ist genau dann überdeckt, falls das zugehörige nicht-negierte bzw. negierte Prädikat mindestens einmal aufgrund der Testmenge zu **`wahr`** *evaluiert.*

Betrachtet man nicht nur einzelne Graphersetzungsregeln sondern ganze Operationsimplementierungen oder auch vollständige Klassen mit mehreren Operationen, so lassen sich die Definitionen der Überdeckungsmaße entsprechend auf die Gesamtheit aller relevanter Coverage-Items erweitern. Wichtig ist, dass dabei der Quotient aus der Gesamtsumme der *überdeckten* Coverage-Items und der Gesamtsumme der generierten Coverage-Items gebildet wird. Rechnerisch ist der resultierende Überdeckungswert dann allerdings nicht mehr als Arithmetisches Mittel der Einzelwerte zu bestimmen.

7.2.3 Ableitung der Coverage-Items

Im Folgenden werden die Details der Umsetzung des in Abbildung 7.2 dargestellten Ansatzes vorgestellt. Es stehen zwei Hauptaspekte im Fokus der Betrachtung, nämlich zum einen die benötigte Ableitungsvorschrift zur Ableitung der *RPs* auf Basis von Metamodellen und *SDM*-Diagrammen in Form des grundlegenden Kontrollalgorithmus, zum anderen die Beschreibung einer Menge von *RP*-Generierungsstrategien, welche den grundlegenden Algorithmus vervollständigen und konkretisieren. Die Beschreibung der Generierungsstrategien erfolgt in erster Linie natürlichsprachlich und wird stellenweise durch Pseudocode ergänzt.

Algorithmus

In Algorithmus 1 ist der grundlegenden Ablauf der *RP*-Generierung skizziert. Auf Basis der *SUT*-Implementierung, repräsentiert durch die Eingabe $\mathsf{rootPackage}_{SUT}$, was für das einzelne Outermost-Package des *SUT*-Metamodells steht, wird eine Menge RP von *RPs* abgeleitet und als Ausgabe zurückgeliefert. Darüber hinaus wird als weitere Ausgabe ein Hilfsmetamodell $\mathsf{rootPackage}_{RP\text{-}Facility}$ erzeugt, welches im Verlauf der Ausführung ebenfalls befüllt wird.

In Abbildung 7.3 ist der Inhalt eines solchen Hilfsmetamodells als Klassendiagramm skizziert. Dargestellt ist ein Ausschnitt für das Endergebnis nach der *RP*-Generierung (für das laufende Beispiel). Die generierten Klassen definieren die Klassen des Datenmodells zur Erfassung von Statistiken für die Berechnung der Überdeckungsgrade. Die Namen solcher Klassen enden mit dem Suffix „TesterStats“. Darüber hinaus beinhaltet das Metamodell eine weitere Sorte von Klassen, deren Namen auf dem Suffix „Tester“ enden. Sie beinhalten Operationen zur Auswertung der generierten *RPs* sowie zur Erfassung und Aktualisierung der dabei anfallenden Überdeckungsmesswerte. Das Namenskonvention bzgl. der Operations- und Attributnamen dieser Klassen sorgt dafür, dass einige zentrale Informationen zum Ursprung dieser in den Bezeichnern kodiert sind. Der Name einer Operation folgt dabei dem Aufbau: <Name der zugrunde liegenden Operation des *SUT*>___<Name des Story-Pattern>___<*RPs*-ID>___<Details zur Erzeugungsstrategie>. Der Name eines Attributs zeigt den gleichen Aufbau, kann aber zusätzlich noch den Präfix „not_“ aufweisen. Letzter deutet an, dass es sich um ein Feld für das Speichern der Überdeckungsinformation eines negierten Prädikats handelt. Ist der Präfix vorhanden, zeigt das Feld an, ob das zugehörige *RP* auch mindestens einmal zu *keinem* Match geführt hat.

Algorithmus 1 : RP-Generierung

```
   Input  : rootPackage_SUT
   Output : RP, rootPackage_SUT, rootPackage_RP-Facility
 1 RP ← ∅;
   /* Erstellen und Initialisieren des RP-Metamodells                    */
 2 initialize(rootPackage_RP-Facility);
   /* RP-Generierung anhand der vorhandenen Story-Patterns               */
 3 CLASSES ← collectAllClassesFrom(rootPackage_SUT);
 4 foreach cl ∈ CLASSES do
       /* Erzeuge 2 Klassen im RP-Metamodell (RP- und Statistiken-Container) */
 5     tc ← createAndAddTesterClassFor(cl, rootPackage_RP-Facility);
 6     tsc ← createAndAddStatsContainerClassFor(cl, rootPackage_RP-Facility);
 7     OPERATIONS_cl ← getAllOperationsOf(cl);
 8     foreach op ∈ OPERATIONS_cl do
 9         activity ← getCorrespondingSdm(op);
10         STORY_NODES ← getAllStoryNodes(activity);
11         foreach sn ∈ STORY_NODES do
               /* Eigentliche Berechnung der RPs                         */
12             RP* ← generateRPsFor(sn);
13             RP ← RP ∪ RP* ;
               /* Ergänzung des RP-Metamodells                           */
14             generateNewStatsVariablesFor(RP*, tsc);
15             EVAL_STATEMENTS ← generateEvaluationStatementsFor(RP*, tc, tsc);
               /* Instrumentierung der originalen Transf. zur RP-Auswertung */
16             addInstrumentationCommand(activity, sn, EVAL_STATEMENTS);
17         end
18     end
19 end
```

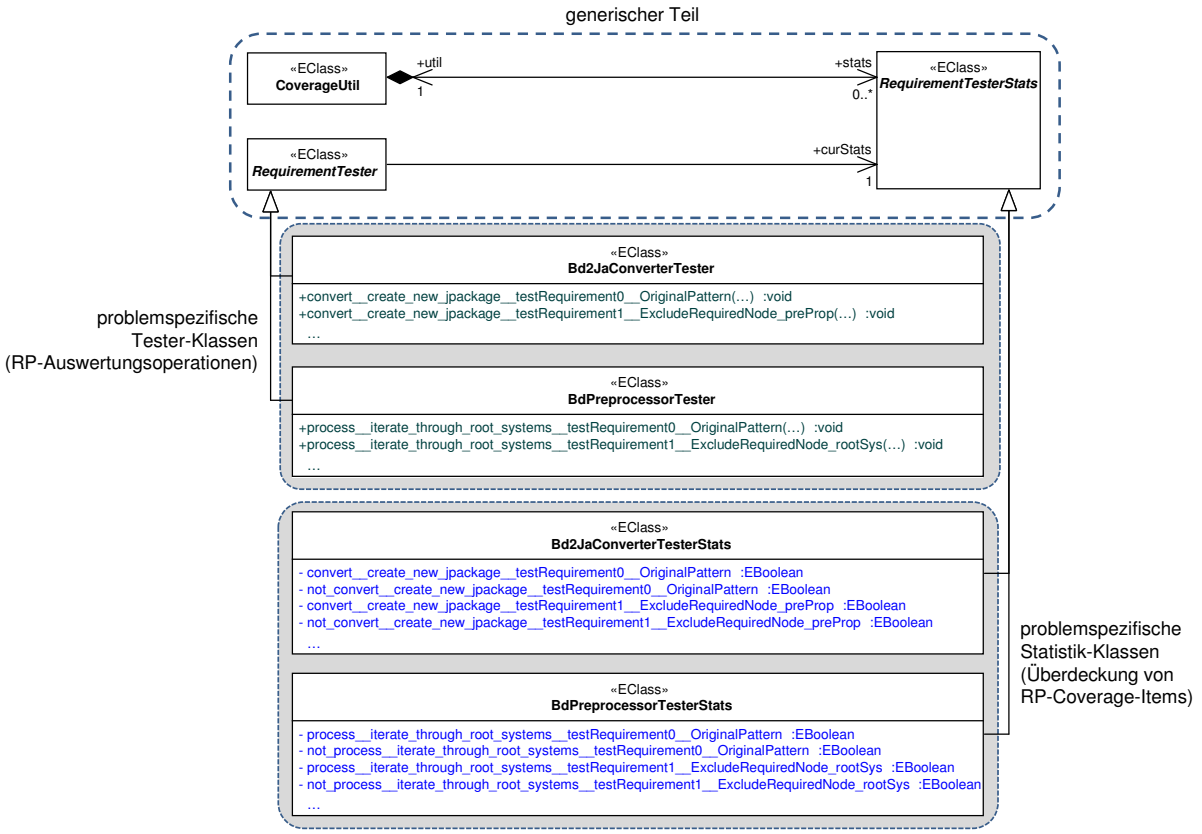


Abbildung 7.3: Generiertes Hilfsmetamodell für Statistiken und *RP*-Auswertungsoperationen

Darüber hinaus wird durch den Algorithmus auch das ursprüngliche *SUT*-Metamodell – genauer, die darin enthaltenen *SDM*-Beschreibungen – angepasst und als weiteres Resultat mit ausgegeben. Die im *SUT*-Metamodell hinterlegten Transformationsimplementierungen werden hierbei für das Testen *instrumentiert*, also um Anweisungen zur Auswertung der *RPs* ergänzt, vgl. 7.2 Schritt ⑥.

Die grundsätzliche Funktionsweise von Algorithmus 1 ist wie folgt: Nachdem in Zeile 1 die *RP*-Menge und in Zeile 2 das Hilfsmetamodell $\mathsf{rootPackage}_{RP\text{-}Facility}$ initialisiert werden, werden in Zeile 3 alle Klassen aus dem Metamodell zusammengesucht und in der Menge CLASSES abgelegt. In den Zeilen 4 bis 20 werden anschließend die einzelnen Klassen der Reihe nach betrachtet. Für jede Klasse des ursprünglichen Metamodells werden im Hilfsmetamodell, vgl. auch Abbildung 7.3, jeweils eine *Tester*-Klasse und eine *TesterStats*-Klasse erzeugt. Die Tester-Klasse enthält im weiteren Verlauf, vgl. Zeile 16, eine Operation für jedes erzeugte *RP*, die das entsprechende *RP* enthält, welches durch einen Aufruf der entsprechenden Operation auswertet werden kann. Die TesterStats-Klasse erhält für jedes erzeugte *RP* eine Boolesche Variable, s. Zeile 15, deren Wert anzeigt, ob das entsprechende Coverage-Item bereits abgedeckt ist. Darüber hinaus werden innerhalb der TesterStats-Klasse Update-Operationen für die Werte der Booleschen Variablen generiert, die bei noch fehlender Überdeckung an die jeweilige Tester-Klasse delegieren, und dadurch nachprüfen, ob das entsprechende Coverage-Item zum aktuellen Zeitpunkt gegebenenfalls abgedeckt ist.

In Zeile 7 werden alle Operationen[1] der aktuellen Klasse gesammelt und die Menge $\mathsf{OPERATIONS}_{cl}$ entsprechend befüllt. Für jede Operation, vgl. die Schleife zwischen den Zeilen 8 bis 19, werden dann die jeweilige *SDM*-Spezifikation (Zeile 9) sowie die darin enthaltenen *Story*-Knoten, die den Graphregeln entsprechen, bestimmt (Zeile 10). Für jeden Story-Knoten werden danach in Zeile 12 entsprechende *RPs* abgeleitet und Zeile 13 dem Gesamtergebnis hinzugefügt. Da sich die wesentlichen Leistungsdaten des Überdeckungsansatzes aus der Generierung der *RPs* ergeben und diese von zentraler Bedeutung sind, wird dieser Teilschritt im weiteren Verlauf separat betrachtet, vgl. hierzu die Beschreibung der hervorgehobenen Hilfsfunktion generateRPsFor im Anschluss.

Nachdem für die betrachtete Operation die Coverage-Items abgeleitet sind, muss die ursprüngliche Implementierung noch so angepasst werden, dass die Überdeckung zur Ausführungszeit auch gemessen wird. Hierzu werden im Rahmen der in Zeile 17 aufgerufenen Hilfsfunktion Instrumentierungsanweisungen in die ursprüngliche *SDM* eingebaut, die dazu führen, dass vor der Auswertung der ursprünglichen Graphtransformationsregel die Menge der dazugehörigen *RPs* ausgewertet und die Statistiken entsprechend aktualisiert werden.

[1] Hierbei werden nur Operationen berücksichtigt, die in *SDM*-Form implementiert sind. (Die Menge beinhaltet auch diejenigen Operationen, die von den Oberklassen vererbt werden.)

Funktion generateRPsFor

```
   Input  : sn /* ein Story-Node                                              */
   Output : RP* /* Menge der generierten PRs                                  */

   /* Initialisierung                                                         */
 1 sp ← sn.storyPattern;
 2 LV ← sp.linkVariables;
 3 OV ← sp.objectVariables;
 4 this ← NIL /* Spezielle „this"-Variable                                   */
 5 if ∃ ov ∈ OV: ov.name = "this" then this ← ov;

   /* Hilfsstruktur „params" befüllen                                         */
 6 params.LV ← LV;
 7 params.OV ← OV;
   /* Teilmengen der Link- und Object-Variablen (mit bestimmten Eigenschaften) */
 8 params.LV_NAC ← {lv ∈ LV | lv has „NEGATIVE“ binding semantics};
 9 params.LV_opt,destr ← {lv ∈ LV | lv has „OPTIONAL“ binding semantics and „DESTROY“ operator};
10 params.LV_opt,!chk← {lv ∈ LV | lv has „OPTIONAL“ binding semantics and not „CHECK_ONLY“
   operator};
11 params.LV_mand,!cr ← {lv ∈ LV | lv has „MANDATORY“ binding semantics and not „CREATE“ operator};
12 params.OV_NAC ← {ov ∈ OV | ov has „NEGATIVE“ binding semantics};
13 params.OV_NAC,ub ← {ov ∈ OV | ov has „NEGATIVE“ binding semantics, and „UNBOUND“ binding
   state};
14 params.OV_opt,ub ← {ov ∈ OV | ov has „OPTIONAL“ binding semantics, and „UNBOUND“ binding
   state};
15 params.OV_opt,ub,!cr ← {ov ∈ OV | ov has „OPTIONAL“ binding semantics, „UNBOUND“ binding state,
   and non-„CREATE“ operator};
16 params.OV_mand,ub,!cr ← {ov ∈ OV | ov has „MANDATORY“ binding semantics, „UNBOUND“ binding
   state, and non-„CREATE“ operator};
17 params.OV_ub,!cr ← (params.OV_NAC,ub ∪ params.OV_opt,ub,!cr ∪ params.OV_mand,ub,!cr);

   /* Vorbereitungsschritte zur Extraktion des rp-Skeletts aus dem sp         */
18 PREP ← createPreparationSteps(sp, this);

19 rp ← cloneStoryPattern(sp);
20 foreach p in PREP do
21 |   execute p on rp;
22 end

   /* Menge von Modifikationen, die das rp-Skelett zum jeweiligen rp machen   */
23 MOD ← createModifications(sp, this, params);

24 foreach m ∈ MOD do
25 |   rp' ← cloneStoryPattern(rp);
26 |   apply m to rp';
27 |   RP* ← RP* ∪ {rp'};
28 end
```

Die Funktion generateRPsFor bildet für einen gegebenen Story-Knoten anhand dessen *GT*-Regel eine Menge von Graphmustern: die Menge der *RPs*. Dazu wird in Zeile 1 aus dem Story-Knoten, einem Knoten des *SDM*-, die eigentliche *GT*-Regel extrahiert. Anschließend werden in den Zeilen 2 und 3 die *OVs* und *LVs* der Regel bestimmt und in den Mengen LV und OV zwischengespeichert. Wie bereits in Kapitel 4 erläutert, stellt die *this-OV* einen Sonderfall unter allen *OVs* dar; sie repräsentiert stets das Objekt aus dem Modell, auf welchem die Operation aufgerufen wurde. Aufgrund dieser Sonderrolle muss sie auch bei der Erzeugung der *RPs* gesondert betrachtet werden. Folglich wird in den Zeilen 4 und 5 diese Spezial-*OV* gesucht und die Variable this entsprechende initialisiert.

Im weiteren Ablauf werden einige spezifische Untermengen der *OV*- und *LV*-Mengen benötigt; sie werden in den Zeilen 6 bis 17 bestimmt. Sie dienen der Übersichtlichkeit der folgenden Darstellung, da so vermieden werden kann, dass wiederholt über die Gesamtheit aller *OVs* bzw. *LVs* iteriert werden muss, was jeweils ein Ausfiltern von nicht unpassenden Elementen erfordern würde. Andererseits wird durch die Einführung des Parameterobjekts (params) erreicht, dass lange Parameterlisten in der weiteren Darstellung benötigt werden.

Die einzelnen Untermengen werden aufgrund der Binding-Eigenschaften der Variablen gebildet, vgl. hierzu insbesondere auch noch mal Abschnitt 4.3.2 bzw. Abbildungen 4.4 und 4.5. So umfasst die Menge $\mathsf{LV}_{\mathrm{NAC}}$ aus Zeile 8 beispielsweise alle *LVs*, die so definiert wurden, dass sie eine *NAC* bilden und ihre Binding-Semantik folglich den Wert „`NEGATIVE`“ trägt. Eine zweite Menge, die *LVs* gruppiert, ist die in Zeile 9 definierte Menge $\mathsf{LV}_{\mathrm{opt,destr}}$. Sie enthält nur solche *LVs*, die als optional spezifiziert sind. Zusätzlich müssen die enthaltenden *LVs* aber auch noch den Binding-Operator „`DESTROY`“ tragen, was bedeutet, dass sie an optional vorhandene Verbindungen gebunden werden und diese, falls vorhanden, löschen. Eine dritte *LV*-Menge wird in Zeile 10 gebildet. Sie enthält sämtliche als optional konfigurierte *LVs* des originären Musters. Optional nur zu überprüfende Variablen sind hierbei ausgeschlossen, da ihre grundsätzliche Sinnhaftigkeit fraglich ist. In Zeile 11 wird die Teilmenge der *LVs* gebildet, die als „`MANDATORY`“ spezifiziert wurden, es dabei aber nicht um zu erzeugende Links geht.

Analog zu $\mathsf{LV}_{\mathrm{NAC}}$ enthält die Menge $\mathsf{OV}_{\mathrm{NAC}}$ aus Zeile 12 nur die Object-Variablen des Story-Pattern, die einen auszuschließenden Knoten im Modellgraphen repräsentieren, und somit auch für diese gilt, dass die Binding-Semantik jeweils den Wert „`NEGATIVE`“ trägt. In Zeile 13 wird die Menge $\mathsf{OV}_{\mathrm{NAC}}$ noch um die gebundenen *OVs* reduziert und das Ergebnis in der Menge $\mathsf{OV}_{\mathrm{NAC,ub}}$ zwischengespeichert. Mit der Definition der Menge $\mathsf{OV}_{\mathrm{opt,ub}}$ in Zeile 14 werden alle ungebundenen *OVs* zusammengefasst, die optionale Binding-Semantik besitzen. Hierbei handelt es sich folglich um alle optionalen Knoten sowie die bedingt anzulegenden und bedingt zu löschende Knoten. In Zeile 15 wird die Menge $\mathsf{OV}_{\mathrm{opt,ub,!cr}}$ eingeführt, welche die *OVs* enthält, die als ungebunden und optional spezifiziert wurden, und darüber hinaus auch nicht bedingt erzeugt werden sollen. Dabei ist es egal, ob der Binding-Operator den Wert „`DESTROY`“ oder „`CHECK_ONLY`“ trägt, was bedeutet, dass entsprechende Modellelemente als Teil der *LHS* der Regel bei der Auswertung der Regelvorbedingung bereits gesucht werden. Die Definition der Menge $\mathsf{OV}_{\mathrm{mand,ub,!cr}}$ erfolgt in Zeile 16 hierzu äquivalent, bis auf die Tatsache, dass statt optionalen die zwingend vorausgesetzten *OVs* enthalten sind. Die Vereinigungsmenge, gebildet aus den drei zuvor beschriebenen Mengen $\mathsf{OV}_{\mathrm{NAC,ub}}$, $\mathsf{OV}_{\mathrm{opt,ub,!cr}}$ und $\mathsf{OV}_{\mathrm{mand,ub,!cr}}$, ergibt zuletzt noch die Menge $\mathsf{OV}_{\mathrm{ub,chk}}$. Sie umfasst *alle ungebundenen OVs*, die zur *LHS* der Regel beitragen.

Nachdem die Variablen des Story-Patterns anhand wichtiger Eigenschaften auf die Teilmengen verteilt sind, werden ab Zeile 18 die *RPs* erzeugt. Eine weitere Grundvoraussetzung ist die *LHS* der betrachteten Regel. Technisch lässt sich diese aus der Regel „extrahieren“, indem aus der Regel alle schreibenden Anteile entfernt bzw. diese so modifiziert werden, dass anschließend nur noch der seiteneffektfreie Musteranteil übrig bleibt. Neu anzulegende Knoten und Kanten sowie Attributmanipulationen werden folglich entfernt. Zu löschende Elemente (Binding-Operator=„`DESTROY`“) beispielsweise werden zu reinen Vorbedingungen umgewandelt (Binding-Operator=„`CHECK_ONLY`“).

Attribut*bedingungen* können grundsätzlich erhalten bleiben, da sie sich auf die linke Seite der *GT*-Regel beziehen. Allerdings können praktische Überlegungen auch dafür sprechen, dass solche Bedingungen immer auch entfernt werden. So können sich Bedingungen beispielsweise auf außerhalb der aktuellen Regel gebundene *OVs* oder deren Attribute beziehen. Diese müssten dann im Rahmen der Auswertung des *RPs* in dem jeweiligen Auswertungskontext bekannt sein. Da die *RPs* in Operationen einer Hilfsklasse eingebettet werden, würde dies eine Erweiterung der Operationsparameter der *RP*-auswertenden Operation erfordern, um so die benötigten Informationen übergeben zu können. Die hierbei benötigten *OVs* müssen zuvor erst durch eine komplexere Analyse aller vorhandenen Attributbedingungen identifiziert werden.[2]

Ein weiterer notwendiger Teilschritt, welcher der Design-Entscheidung zur Auslagerung der *RP*-Auswertung in Hilfsklasse geschuldet ist, besteht darin, die spezielle *this*-Variable, falls sie denn Teil der *LHS* der Regel ist, innerhalb der *RPs* umzubenennen,[3] um aus ihr eine „normale“ (gebundene) *OV* zu machen. Da sich die *this*-Variable stets auf die Laufzeitinstanz bezieht, auf welcher die *SDM*-Operation aufgerufen wird, würde sich die Variable ansonsten fälschlicherweise auf eben jene Hilfsklasse (mit falschem Typ) beziehen und nicht auf die ursprüngliche Klasse, für welche die *SDM*-Transformation implementiert wurde. Damit die neu eingeführte Ersatzvariable den richtigen Wert referenziert, muss die Signatur der Auswertungsoperation so angepasst werden, dass der eigentliche Wert der ursprünglichen *this*-Variablen über den neuen Parameter an die *RP*-auswertende Operation übergeben werden kann. Die notwendigen Anpassungen werden in anderen Hilfsfunktionen von Algorithmus 1 vorgenommen.

Alle Schritte die nötig sind, um aus der ursprünglichen Graphregel die *LHS* und damit das prototypische *RP*-„Gerüst“ abzuleiten, werden in Zeile 18 der Hilfsfunktion `createPreparationSteps(.)` in Form von Modifikationsanweisungen erzeugt, welche anschließend in der Menge PREP verwaltet werden. Auf eine dedizierte Betrachtung jener Hilfsfunktion soll hier verzichtet werden, da die durchzuführenden Modifikationen auf konzeptioneller Ebene bereits beschrieben wurden. Vor der Durchführung der eigentlichen Anpassungen aus PREP wird in Zeile 19 ein Rohling des *RP* durch vollständiges Klonen der ursprünglichen Regel, inkl. der schreibenden Anteile, erzeugt und in der Variable rp abgelegt. In der nachfolgenden Schleife (Zeilen 20 bis 22) wird das rudimentäre *RP* schrittweise zum ersten Basis-*RP* vereinfacht, indem die notwendigen Operationen ausgeführt werden. Nach der Zeile 22 repräsentiert rp die *LHS* der *GT*-Regel.

Um, ausgehend von dem in rp gespeicherten Basismuster, weitere *RPs* abzuleiten, müssen systematisch weitere Teile des Musters manipuliert werden, um so zusätzliche

[2] Die im Rahmen dieser Arbeit genutzte Implementierung leistet dies in der umgesetzten Ausbaustufe noch nicht. Folglich werden Attributbedingungen komplett entfernt.

[3] In der Implementierung wird der Name der Variable von „this“ zu „myThis“ geändert.

Anforderungen an die Testmodelle stellen zu können, als dass nur das ursprüngliche Muster einmal gefunden wird und einmal nicht gefunden wird. Z. B. soll der Typ einer *OV* verändert oder einzelne Variablen des Musters ganz entfernt werden können. Art, Anzahl und Anwendungsstellen solcher Änderungsschritte bestimmen unmittelbar, welche Coverage-Items entstehen und anschließend durch das Testen überdeckt werden müssen. Somit bestimmen entsprechende Strategien mittelbar die zu erwartenden und tatsächlichen Eigenschaften des Testansatzes. Konkrete Strategien, die hierfür genutzt werden können, werden im unmittelbar nachfolgenden Teilabschnitt unter Bezug auf die Funktion createModifications beschrieben. Letztere wird in Zeile 23 der Funktion generateRPsFor aufgerufen, und die zurückgelieferte Menge atomarer Modifikationen wird in MOD verwaltet.

Im letzten Teil der Funktion generateRPsFor wird zwischen den Zeilen 24 bis 28 genau ein entsprechendes *RP* anhand einer jeden atomaren Modifikation m abgeleitet. Dafür wird die jeweilig notwendige Modifikation auf eine Kopie des Basis-*RPs* ausgeführt und das resultierende *RP*, hier als rp' bezeichnet, wird der Endergebnismenge RP* hinzugefügt. Danach endet der Teilablauf und die Menge RP* wird an den Aufrufer übergeben.

RP-Generierungsstrategien

Zur Beschreibung der Strategien zur Ableitung von *RPs* aus einer Regel betrachten wir nun die Funktionsweise und den inneren Aufbau der im Folgenden gegebenen Hilfsfunktion createModifications genauer. Aufgerufen wird sie in Zeile 23 der Hilfsfunktion generateRPsFor. Das generelle Vorgehen basiert auf der Modifikation der *LHS* und weist damit, wie bereits in Abschnitt 6.4.1 ausgeführt, gewisse Ähnlichkeiten zum Ansatz von Darabos et al. [DPV08] auf.

Insgesamt sind die im Folgenden beschriebenen Modifikationen „defensiver“ Natur, d. h. sie ändern das Muster möglichst nur so, dass sichergestellt ist, dass das Ergebnis auch noch valide und erfüllbar ist. *RPs*, die leicht als nicht erfüllbar zu erkennen sind, sollten möglichst schon nicht generiert werden. Dies ist ein Unterschied zum Mutationstesten, da dort diese Einschränkung nicht notwendig bzw. nicht erwünscht ist. Es sollen in letzterem Fall einfach Fehler beim Editieren simuliert werden, die durchaus auch zu fehlerhaften Spezifikationen führen können. Solche fehlerhaften Mutanten lassen sich auch tendenziell leicht identifizieren.

Unveränderte Kopie (UC - Unmodified Copy). Die *UC*-Strategie stellt sicher, dass das ursprüngliche Muster aus der Regel entweder komplett unverändert oder zumindest so weit wie möglich semantikerhaltend übernommen wird. Wie bereits zuvor erwähnt, kann es hierfür unter Umständen notwendig sein, dass eine `this`-*OV* in eine normale *OV* konvertiert wird. Das Objekt, an das die abgewandelte *OV* gebunden wird, bestimmt sich nach der Anpassung nicht mehr über die Instanz, auf der die Operation aufgerufen wird, sondern anhand eines Aufrufparameters der Operation. In der Funktion createModifications wird diese Strategie in den Zeilen 1 bis 5 umgesetzt. Die Konstante NULL_MODIFICATION steht dabei für keine tatsächliche Modifikation, sondern sagt lediglich aus, dass für dieses Element keine Änderungen am Muster vorgenommen werden sollen. Sie wird in Zeile 2 zur Menge der Modifikationen hinzugefügt, falls `this` nicht vorhanden ist. Sollte `this` vorhanden sein, liefert die Hilfsfunktion createRenameThisOv in Zeile 4 eine andere Modifikation, welche die Umbenennung der *OV* veranlasst.

Funktion createModifications

```
   Input  : sp, this, params /* Story-Pattern, „this"-OV, Parameterobjekt          */
   Output : MOD /* Menge von Modifikationen                                        */
 1 if this = NIL then
 2 |  MOD ← {NULL_MODIFICATION}; /* Initialisierung mit neutralem Element        */
 3 else
 4 |  MOD ← {createRenameThisOv(this)}; /* Umbenennen von „this"-OV              */
 5 end
 6 MOD ← MOD ∪ createRequireAllNacsMod(params.OV_NAC, params.LV_NAC);
 7 MOD ← MOD ∪ createRemoveAllNacsMod(params.OV_NAC, params.LV_NAC);
 8 typeHelper ← getTypeHelper();
 9 foreach ov ∈ params.OV_ub,!cr do
   |  /* OV-Typ auf kompatiblen Typ ändern                                        */
10 |  MOD ← MOD ∪ {createAlterOvTypeMod(ov,typeHelper,...)};
11 end
12 foreach ov ∈ params.OV_opt,ub do
   |  /* optionale OV fordern, optionale OV ausschließen                         */
13 |  MOD ← MOD ∪ {createRequireOptionalOvMod(ov,...)};
14 |  MOD ← MOD ∪ {createExcludeOptionalOvMod(ov,...)};
15 end
16 foreach ov ∈ params.OV_mand,ub,!cr do
   |  /* mandatory OV als NAC ausschließen (Zusammenhang beachten)               */
17 |  MOD ← MOD ∪ {createSwitchOvToNacMod(ov,...)};
18 end
19 foreach lv ∈ params.LV_mand,!cr do
   |  /* mandatory LV als NAC ausschließen (Zusammenhang beachten)               */
20 |  MOD ← MOD ∪ {createSwitchLvToNacMod(lv,...)};
21 end
22 foreach lv ∈ params.LV_opt,!chk do
   |  /* optional zu löschende LVs fordern und ausschließen                      */
23 |  MOD ← MOD ∪ {createRequireOptionalLvMod(lv,...)};
24 |  MOD ← MOD ∪ {createExcludeOptionalLvMod(lv,...)};
25 end
26 foreach lv ∈ params.LV do
   |  /* LV-Typen durch kompatiblen Typen ersetzen                              */
27 |  MOD ← MOD ∪ {createSimpleAlterLvTypeMod(lv,...)};
28 |  MOD ← MOD ∪ {createComplexAlterLvTypeMod(lv,...)};
29 end
```

Alle NAC-Elemente fordern (RAN - Require All NACs). Die *RAN*-Strategie stellt sicher, dass alle *NAC-OVs* bzw. *LVs* des aktuell betrachteten Musters dahingehend modifiziert werden, dass ihre jeweilige *Binding-Semantik* von `NEGATIVE` auf `MANDATORY` abgeändert wird. Es werden alle *NAC*-Elemente gleichzeitig gefordert, da so sichergestellt ist, dass das ursprüngliche Muster potentiell auf eine Konstellation im Modell trifft, die aufgrund der ersten *NAC*-Überprüfung ausgeschlossen werden kann. Entsprechende Modifikationen werden in Zeile 6 der createModifications-Funktion angelegt. Weitere Strategien könnten sich auch auf einzelne *NAC*-Elemente beschränken.

Diese Anpassung des Musters lässt sich im Allgemeinen weder als *Verschärfung* noch als *Aufweichung* auffassen: Zwar sind einerseits die ursprünglichen Vorbedingungen, die

sich aufgrund der *NAC*-Elemente ergeben, nach der Ausführung einer solchen Modifikation obsolet, d.h. das Muster wurde in dieser Hinsicht aufgeweicht, da es ohne die *NAC*-Elemente theoretisch auf mehr Modellen zu einem Match führen würde. Dafür ergeben sich andererseits allerdings neue, veränderte Vorbedingungen durch die Umwandlung der Binding-Semantik. Die früheren *NAC*-Elemente werden ja anschließend zwingend vorausgesetzt, was man wieder als Verschärfung des Musters auffassen kann.

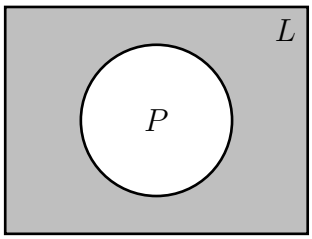
Anhand der Abbildung 7.4 lässt sich dies leicht nachvollziehen: Ein einfaches Muster P ohne *NACs*, vgl. Abbildung 7.4a, wird durch zusätzliche Variablen zu P' verschärft, so dass sich eine kleinere Menge an Modellen ergibt, auf denen das Muster zu Treffern führen würde. Die sich durch das verschärfte Muster P' ergebende Situation ist in Abbildung 7.4b skizziert. Werden jetzt allerdings die zusätzlichen Forderungen von P' gegenüber P als *NACs* konfiguriert, ergibt sich die in Abbildung 7.4c dargestellte Situation. Das neue Muster P'' mit *NACs* führt also dann zu Treffern, wenn P matchen und P' nicht matchen würde. Folglich resultiert die von der *RAN*-Strategie geforderte Modifikation darin, dass statt dem ursprünglichen Muster, welches P'' aus Abbildung 7.4c entspricht, im Anschluss ein *RP* erzeugt wird, welches sich zum ursprünglichen Muster wie P' aus Abbildung 7.4b zu P'' verhält.

Alle NACs entfernen (REMAN - Remove All NACs). Analog zur *RAN*-Strategie werden bei der *REMAN*-Strategie ebenfalls die *NAC*-Elemente betrachtet. Allerdings sorgt diese Strategie dafür, dass eine solche Modifikation in Zeile 7 erzeugt wird, die alle *NAC*-Elemente ganz aus dem prototypischen Muster entfernt, so dass am Ende nur der Teil ohne *NACs* bestehen bleibt.

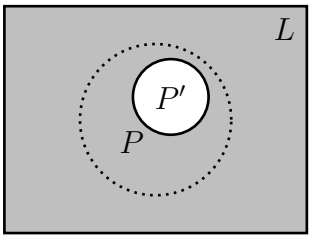
Auch diese Anpassung lässt sich mit Hilfe der Abbildung 7.4c nachvollziehen. Das ursprüngliche Muster, das auch hier P'' in der Ausgangssituation entspricht, wird durch die Anwendung der Modifikation zu einem *RP*, das in seinen Forderungen dem Muster P aus Abbildung 7.4a entspricht. Testmodelle, auf denen das so erzeugte *RP keinen* Treffer findet, unterscheiden sich ggf. grundsätzlich von solchen Modellen, welche keinen Treffer für das ursprüngliche Muster enthalten.

Ändern des OV-Typs (COT - Change Object variable Types). Jede *OV* besitzt einen *nominalen* Typ, welcher einer konkreten oder abstrakten Klasse aus dem zugrunde liegenden Metamodell entspricht. Dieser wird durch den Entwickler beim Anlegen der *OV* in der Beschreibung explizit festgelegt. Wie bereits in Abschnitt 4.3.2 erläutert, kann eine *OV* zur Laufzeit nur an solche Objekte aus dem Modell gebunden werden, deren tatsächlicher Typ zu diesem *OV*-Typen *kompatible* ist. Folglich muss ein kompatibles Objekte entweder Instanz der besagten Klasse selbst sein oder von einer konkreten Unterklassen derselben.

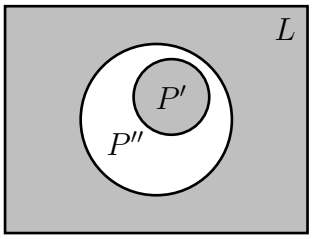
Das Ziel der *COT*-Strategie besteht darin, die *GT*-Regeln des *SUT* möglichst auch mit solchen Modellen zu testen, in denen Instanzen jeder der relevanten Unterklassen an den entscheidenden Stellen vorhanden sind. Damit soll überprüft werden, dass sich die Regel in allen der möglichen Konstellationen korrekt verhält und sich ggf. auch tatsächlich auf entsprechenden Strukturen als anwendbar erweist. Darüber hinaus ist ebenfalls von Interesse, wie sich die Regel in solchen Fällen verhält, in denen das Muster *fast* vorliegt, und es nur deshalb nicht zu einem Treffer auf dieser Teilstruktur kommt, da ein hierfür benötigtes und vorhandenes Objekt im Modell vom falschen Typ ist. Der Begriff des



(a) Einfaches Muster P ohne *NACs*



(b) Muster P' durch zusätzliche Forderungen der ursprünglichen *NAC*-Elemente gegenüber P



(c) Zusätzliche Forderungen in *NAC*-Form führen von P zu P'' (weißer Bereich)

Abbildung 7.4: Veranschaulichung der *RAN*- und der *REMAN*-Strategien

„falschen Typs“ bezieht sich konkret darauf, dass ein Objekt im Modell eine Instanz einer zu allgemeinen, inkompatiblen Oberklasse, statt der eigentlich benötigten Unterklasse ist. Beide möglichen Varianten, also die Betrachtung von spezielleren Unterklassen und inkompatiblen Oberklassen, werden nun separat betrachtet.

Substitution durch Unterklassen Als erstes betrachten wir den Fall, in dem Unterklassen der nominalen *OV*-Typen berücksichtigt werden und Instanzen dieser im Modell durch *RPs* eingefordert werden sollen. Dazu soll sichergestellt werden, dass das Binden in solchen Situationen erfolgt, die in ihrer Gesamtheit sicherstellen, dass jede *OVs* an Objekte aller für sie möglichen realen Typen zumindest hätte gebunden werden können. Einfacher ausgedrückt, es wird gefordert, dass im Laufe des Testens die ungebundenen Variablen beim Binden Objekte von allen für sie jeweils relevanten Typen „sieht“.[4] Die Menge der kompatiblen Typen bei einer *OV* umfasst die Klassen des bereits in Kapitel 2 erwähnten *Vererbungsclans*, vgl. z. B. [BET08], des *OV*-Typs sowie die Klasse der *OV* selbst. Wir bezeichnen die Vereinigungsmenge als den *erweiterten* Vererbungsclan und nutzen im Folgenden die Abkürzung $\text{clan}^*(.)$. Das Gegenstück dazu bildet die Oberklassenmenge $\text{super}(c)$, die alle direkten und indirekten Oberklassen von c enthält.

Nachdem anhand der beiden Mengen, $\text{clan}^*(c)$ und $\text{super}(c)$, nun klar ist, welche Klassen für eine Substitution grundsätzlich in Frage kommen, muss noch erklärt werden, wie man daraus entsprechende *RPs* ableitet und was durch diese ausgedrückt wird. Um zu fordern, dass Instanzen entsprechender Unterklassen in Testmodellen tatsächlich vorhanden sind, wird für eine einzelne ungebundene *OVs* deren ursprünglicher Typ durch den einer Unterklasse ersetzt. Für jede ungebundene *OV* und jede Unterklasse der ursprünglichen Klasse wird also eine Modifikation generiert, die sicherstellt, dass der Typ

[4] Dies lässt sich praktisch natürlich nur durch Instanzen *konkreter* (Unter-)Klassen erreichen, da für abstrakte Klassen keine Instanzen existieren.

entsprechend angepasst wird, und dass für jede solche Änderung ein eigenes *RP* entsteht. Die benötigten Modifikationen entstehen in der Hilfsfunktion `createAlterOvTypeMod` die in Zeile 10 der createModifications-Funktion für jede ungebundene *OV* des Musterteils der *GT*-Regel aufgerufen wird.

Substitution durch Oberklassen Betrachten wir nun den Fall, dass der Typ der *OV* dahingehend geändert werden soll, dass er durch eine seiner Oberklassen ersetzt wird. Es werden auch hier nur ungebundene *OVs* berücksichtigt. Folglich sind alle relevanten *OVs* des ursprünglichen Musters mit anderen Knoten über *LVs* verbunden, da sie ansonsten nicht gebunden werden könnten.[5]

Das hat zur Folge, dass man beim Ersetzen des ursprünglichen Typs durch eine Oberklasse beachten muss, dass die neue Klasse ein geeignetes und *kompatibles* Substitut darstellt. Dies ist nicht immer gegeben, da i. Allg. ein Obertyp gerade *nicht* äquivalent zum Untertyp ist – die Unterklasse kann bekanntermaßen die Oberklasse um Attribute und insbesondere auch Referenzen erweitern. Unmittelbar offenkundig werden potentielle Konflikte, wenn man sich vergegenwärtigt, dass eine Mehrfachvererbung im Metamodell nicht ausgeschlossen ist. Somit ist es beispielsweise leicht möglich, dass eine Klasse zwei Mengen an Referenzen erben kann, und dass, falls eine *OV* über Referenzen aus beiderlei Mengen mit dem restlichen Muster verbunden ist, keine der Oberklassen geeignet ist, die benötigten Eigenschaften in sich zu vereinen. Folglich sollte deutlich geworden sein, dass Elemente der Oberklassenmenge hinsichtlich ihrer jeweiligen Eignung als Substitut zu überprüfen sind.

Zur Bestimmung der kompatiblen Oberklassen müssen die ungeeigneten Kandidaten aus der Menge aller Oberklassen herausgefiltert werden. Hierzu werden die *LVs* des Musters analysiert, die an der betrachteten *OV* enden. Eine *LV* besitzt, wie auch die zugrunde liegende Assoziation im Metamodell, auf konzeptioneller Ebene keine Richtung.[6] Allerdings besitzt jede *LVs* mindestens ein ausgezeichnetes und benanntes Ende, welches per Navigation erreichbar ist und sich im Code als Referenz manifestiert. Die zulässigen Navigierrichtungen der *LVs* werden im Modellierungswerkzeug festgelegt. Hierbei ergibt sich auf technischer Ebene durchaus eine Richtung der Links, nämlich vom Quellende zum Zielende hin, welche durch den Benutzer beim Anlegevorgang bestimmt wird. Im Falle einseitig navigierbarer Assoziationen ist beispielsweise das Quellende der entsprechenden *LVs* bzw. des Links stets *anonym*. Die weitere Darstellung orientiert sich an dieser technischen Sicht, so dass eine Richtung der *LVs* bzw. der jeweiligen Links angenommen werden darf. Somit ergibt sich für jede *OV* eine Menge von eingehenden und ausgehenden Links.

Die Filterung der Oberklassen erfolgt nun zweistufig. Als erstes wird untersucht, ob für einen Oberklassenkandidaten alle benötigten ausgehenden Links (als Referenzen der Oberklasse auf entsprechende Zielklassen) definiert sind. Ist dies nicht der Fall, so ist die Oberklasse ungeeignet und wird aussortiert. Für nicht aussortierte Kandidaten werden anschließend die eingehenden Links überprüft, und zwar dahingehend, ob sie entweder (a) auf Basis einer solchen Assoziation im Metamodell definiert wurden, welche die aktu-

[5] Die der eMoflon zugrunde liegende Modellierungsschicht kann Objekte nur entlang ihrer Beziehungen zu anderen, bekannten Objekten erreichen. Ein Binden von ungebundenen, isolierten *OVs* ohne Kontext ist nicht möglich.

[6] Assoziationen können aber „gerichtet“ im Sinne einer Navigations- oder auch einer Leserichtung sein.

ell betrachtete Oberklasse als Ziel hat, oder (b) die Zielklasse der Assoziation wiederum selbst eine Oberklasse des gerade betrachteten Kandidaten ist. In letzterem Fall ist die betrachtete Oberklasse Teil des Vererbungsclans der Zielklasse der Assoziation und dadurch ein kompatibles Substitut. Trifft keiner der genannten Fälle zu, ist der Kandidat ebenfalls auszusortieren. Die dann verbliebenen Oberklassen stellen bereits kompatible Substitute dar, da Attributbedingungen nicht gesondert analysiert werden müssen. Sie wurden bereits aus anderen Gründen ausgeschlossen, vgl. S. 161. Für jedes valide Substitut werden entsprechende Modifikationen in der Hilfsfunktion `createAlterOvTypeMod`, vgl. den Aufruf in Zeile 10 von createModifications, erzeugt.

Optionale Object-Variable verlangen (ROOV - Require Optional OV). Falls das Muster optionale *OVs* enthält, sollte man testen, wie sich Regel und Transformation verhalten, wenn Entsprechungen solcher optionalen Knoten tatsächlich im Modell vorhanden sind. Eine entsprechende Bedingung lässt sich formulieren, indem man die optionalen Variablen dergestalt umkonfiguriert, dass ein entsprechendes Objekt im Modell als obligatorisch vorausgesetzt und im Falle einer erfolgreichen Auswertung auch gefunden wird. Für bedingt zu erzeugende Knoten muss hierbei beachtet werden, dass der schreibende Aspekt der *OV* entfernt werden muss, denn auch die durch die *ROOV*-Strategie erzeugten *RPs* sollen frei von Seiteneffekten sein. Es mag verwundern, dass optional zu erzeugende Knoten hier überhaupt betrachtet werden, da sich entsprechende *OVs* auf die *RHS* der Regel beziehen. Allerdings sorgt die spezielle Semantik dafür, dass entsprechende Objekte im Modell vor einer Erzeugung erst einmal gesucht werden, um auszuschließen, dass sie bereits vorhanden sind. Diese Suche unterscheidet sich nicht von der Suche nach anderen Objekten der *LHS*, was erklärt, warum diese bedingt zu erzeugenden Knoten ebenfalls berücksichtigt werden.

Entsprechende Modifikationen, die jeweils dafür sorgen, dass *eine* der ursprünglich optionalen *OVs* zwingend im Modell verlangt werden, werden in Zeile 13 der createModifications-Funktion durch Delegation an eine entsprechende Hilfsfunktion generiert. Letztere ist hier ausgespart, da sich in ihre keine wesentliche Komplexität mehr verbirgt.

Optionale Object-Variable ausschließen (EOOV - Exclude Optional OV). Die Negation der Muster, die durch die zuvor beschriebene *ROOV*-Strategie erzeugt wurden, kann nicht garantieren, dass die Suche nach einem Match für die entsprechenden *RPs* aufgrund des Nichtvorhandenseins der ursprünglich optionalen Elemente scheitert. Prinzipiell kämen hierfür auch andere Ursachen in Frage, wie beispielsweise, dass nur Teile der gesuchten Struktur im Modell vorhanden sind. Somit kann durch die aufgrund der *ROOV*-Strategie abgeleiteten Muster nicht garantiert werden, dass auch mit solchen Testmodellen getestet wurde, in der die gesuchte Struktur *ohne die optionalen Knoten* vorhanden sind. Um dies dennoch sicherzustellen, wird durch die *EOOV*-Strategie gefordert, dass zu jeder optionalen *OV* ein *RP* erzeugt wird, in dem diese Variable in *NAC*-Form vorkommt. Dies sorgt dann dafür, dass ein entsprechendes Objekt, im Falle eines Matches des *RP*, im Kontext des identifizierten Teilgraphen *nicht vorhanden* ist. Die entsprechenden Modifikationen, die dies veranlassen, werden in Zeile 14 generiert.

Zwingend notwendigen Knoten ausschließen (EMOV - Exclude Mandatory OV). Das Ziel der *EMOV*-Strategie liegt darin, *RPs* zu generieren, die fordern, dass einzelne ungebundene, obligatorische *OVs* des ursprünglichen Musters nicht an Objekte im Testmodell gebunden werden können. Dies geschieht dadurch, dass eine entsprechende *OV* so umkonfiguriert wird, dass sie statt der ursprünglichen Binding-Semantik `MANDATORY` die Binding-Semantik `NEGATIVE` erhält. Soll ein solches Muster des resultierenden *RP* im Modell gefunden werden, muss ein Teilgraph existieren, der diese neue *NAC* nicht verletzt. Sollte ein passender Teilgraph dann durch die ursprünglichen *GT*-Regeln verarbeitet werden, so stellt dieser Teilgraph aufgrund des fehlenden Objektes – ein solches Objekt wurde ja durch die neu generierte *NAC* explizit ausgeschlossen – keinen vollständigen Match dar. Die Mustersuche muss folglich an dieser Stelle abbrechen und nach einem anderen Treffer im Modell suchen. Aufgrund des Nichtdeterminismus bei der Mustersuche lässt sich allerdings nicht garantieren, dass die ursprüngliche *GT*-Regel auch tatsächlich diesen unvollständigen Teilgraphen berücksichtigt, falls noch ein anderer Match im Modell existieren sollte.

Zur Erzeugung entsprechender *RPs* werden in Zeile 17 der createModifications-Funktion die Modifikationen der *EMOV*-Strategie generiert. Die Modifikationen selbst sind dabei nicht aufwendiger als die zuvor beschriebenen. Allerdings gibt es bei der Erzeugung der Modifikationen einen Aspekt, der gesondert betrachtet werden muss. So weisen viele Muster eine Struktur auf, die es erfordert, dass jeder valide Suchplan bestimmte Knoten immer binden muss, um weitere Konten überhaupt erst erreichen zu können. Es muss sichergestellt sein, dass jeder ungebundene *OV* über einen gültigen Pfad von einem der gebundenen Knoten aus erreichbar ist. Dabei darf ein gültiger Pfad keine *NAC*-Elemente oder zu erzeugende Elemente bzw. optionale Elemente als intermediäre Elemente enthalten. Somit wird ersichtlich, dass nicht einfach beliebige ungebundene, obligatorische *OVs* zu *NACs* abgeändert werden dürfen, da ansonsten andere ungebundene *OVs* ggf. nicht mehr erreichbar sind und so isolierte Teilstrukturen im Muster entstehen können.

Eine Lösung, die dieser Tatsache Rechnung trägt, besteht darin, dass man das Entfernen einer *OV* aus der Menge der für einen Suchplan nutzbaren traversierbaren Knoten zuerst „simuliert“. Dabei kann dann geprüft werden, ob jeder weitere zu bindende Knoten noch zu einer entsprechenden Zusammenhangskomponente gehört, die mindestens einen bereits gebundenen Knoten umfasst und somit einen legalen Suchplan ermöglicht. Es sei darauf hingewiesen, dass für diese Analyse kein Suchplan bestimmt werden muss. Statt dessen interpretiert man das Muster und seine Elemente als einfachen, ungerichteten Graphen mit Knoten und Kanten, und nutzt einfache Graphalgorithmen, um die Frage nach den verlangten Zusammenhangseigenschaften des Graphen zu beantworten.

Ein weiterer Lösungsansatz, der im Rahmen dieser Arbeit allerdings nicht weiter verfolgt wurde, besteht darin, dass man analysiert, welche Elemente durch die Umwandlung einer beliebigen *OV* (mit den benötigten Eigenschaften) anschließend isoliert wären. Diese Elemente könnte man dann gleichfalls aus dem resultierenden *RP* entfernen. Hierdurch würde ebenfalls sichergestellt werden, dass ein gültiger Suchplan existiert. Der Hauptnachteil einer solchen Lösung gegenüber dem zuvor beschriebenen Ansatz liegt im höheren Aufwand – insbesondere was Analyse und Implementierung angeht. Von Vorteil wäre dagegen, dass deutlich mehr *RPs* entstehen würden, falls das Muster viele bzw. lange „verkettete“ Strukturen aus ungebundenen, obligatorischen *OVs* umfasst. Entsprechend würde hierdurch das Testen der Regel durch Modelle mit verschieden großen Teilstrukturen der gesuchten Struktur gefördert.

Zwingend notwendige Kante ausschließen (EMLV - Exclude Mandatory LV). Analog zum Ansatz der *EMOV*-Strategie lassen sich *RPs* auch dadurch erzeugen, dass anstelle von *OVs* obligatorisch vorausgesetzte *LVs* des originären Musters ausgeschlossen werden. Hierbei wird für einzelne *LVs* die Binding-Semantik von `MANDATORY` auf `NEGATIVE` geändert. Dabei ist ebenfalls darauf zu achten, dass keine durch einen Suchplan unerreichbaren Teile im Muster entstehen. Es muss also sichergestellt werden, dass ungebundene Knoten anderweitig auch ohne die dann ausgeschlossene und damit nicht-traversierbare Kante erreichbar bleibt. Das ist bezüglich ungebundener *OVs* grundsätzlich nur dann möglich, wenn jeweils *mindestens ein weiterer* Link existiert, der die jeweilige *OV* mit anderen *OVs* verbindet. Zusätzlich muss gelten, dass diese weiteren Links entweder (a) „parallel“ zum auszuschließenden Link verlaufen, also die selben beiden *OVs* verbinden, oder (b) mindestens einer dieser zusätzlichen Links einen Teil eines gültigen Pfades zu einem gebundenen Knoten darstellt, so dass weiterhin ein gültiger Suchplan möglich ist, der bei der Mustersuche den Teiltreffer grundsätzlich entlang dieser Verbindung vervollständigen kann. In der Funktion createModifications werden die entsprechenden Modifikationen in Zeile 20 generiert.

Optionale Link-Variable verlangen (ROLV - Require Optional LV). Zuvor im Text wurde bereits die *ROOV*-Strategie beschrieben, mit der sichergestellt wird, dass ursprünglich als optional spezifizierte *OVs* in den Testmodellen verlangt und auch mindestens einmal gefunden werden. Selbiges lässt sich ebenfalls für *LVs* im Rahmen der hier betrachteten *ROLV*-Strategie einfordern, mit dem kleinen Unterschied, dass optionalen *LVs* nur dann eine sinnvolle Semantik zukommt, wenn sie auch einen Operator tragen, der *nicht* `CHECK_ONLY` ist.[7] Durch die *ROLV*-Strategie werden *RPs* abgeleitet, in denen einzelne der ursprünglich optionalen *LVs* als obligatorisch vorausgesetzt werden. Die ursprünglich schreibende Semantik der jeweiligen *LV* wird hierzu entfernt, um auch hier wiederum Seiteneffekte auszuschließen. In der Hilfsfunktion werden die entsprechenden Modifikationen in Zeile 23 erzeugt.

Optionale Link-Variable ausschließen (EOLV - Exclude Optional LV). Analog zur *EOOV*-Strategie ist die hier genannte *EOLV*-Strategie als Gegenstück zur *ROLV*-Strategie zu verstehen. Die ursprünglich optionalen *LVs* werden durch Umkonfiguration zu *NAC-LVs*. Sie zeigen beim Auftreten von Treffern für die resultierenden *RPs* im Rahmen des Testens, dass Strukturen im Modell gefunden werden konnten, in denen die ursprünglich nur optional anzulegenden oder zu löschenden Verbindungen *nicht* bzw. noch nicht vorhanden waren. In Zeile 24 wird eine entsprechende Modifikation der Menge der Modifikationen hinzugefügt. Somit ist zumindest die Ausgangssituationen gegeben, dass die zugrunde liegende Regel der *SDM*-Transformation entweder einen nicht vorhandenen, optional zu erzeugenden Link anlegen könnte respektive einen nicht vorhandenen Link gar nicht erst löschen müsste. Dass einer diesen beiden Fälle tatsächlich zum Tragen kommt, kann allerdings wieder nicht garantiert werden, da andere Treffer in den Modellen hier nicht ausgeschlossen sind, welche entsprechende Links doch aufweisen. Um solche Fälle sicher auszuschließen und nur die angestrebte Situation zu erlauben, müsste man einzelne *EOLV-RPs* mit einem zweiten Muster zu einer *SDMs* mit komplexerem Auf-

[7] Dies bedeute, dass die an sie gebundenen Referenzen im Modell entweder bedingt gelöscht oder bedingt erzeugt werden sollen.

bau kombinieren. Die in Abbildung 7.2 im linken unteren Bereich angedeuteten Struktur reicht hierfür nicht aus.

Einfaches Anpassen des Link-Typs (CLTS - Change Link variable Type Simple). Bei der Verknüpfung zweier *OVs* in einer *GT*-Regel mittels einer *LV* kann ein Entwickler in Abhängigkeit vom Metamodell potentiell zwischen mehreren möglichen Link-Typen wählen. Selbstverständlich ist zwischen den Optionen mit semantischen oder auch syntaktischen[8] Unterschieden zu rechnen. Dennoch ist es nie ganz ausgeschlossen, eine ungewollte bzw. falsche Option zu wählen, ohne dass der Fehler unmittelbar erkannt wird. Um den Entwickler im Rahmen des Testens dazu zu bringen, sich entsprechender Entscheidungen noch einmal bewusst zu werden und diese ggf. zu korrigieren, werden durch die *CLTS*-Strategie *RPs* generiert, in denen jeweils eine der obligatorischen *LVs* zu einer *NAC* umkonfiguriert und durch eine neue *LV* eines anderen Typs ersetzt wird.

Die Strategie sorgt so dafür, dass Strukturen in den Testmodellen eingefordert werden, bei denen, statt der ursprünglich verlangten Kante, eine Kante von einem anderen, zu den beteiligen *OVs* kompatiblen Typ vorhanden ist. In Abbildung 7.5 ist ein entsprechendes Beispiel zur Veranschaulichung dargestellt. Das konkrete Beispiel basiert auf Teilen der Operation `collectRelevantBlocks` aus Abbildung A.4. Auf der linken Seite des Pfeils ist das Ausgangsmuster zu sehen. Rechts ist das *RP* gezeigt, das anhand der *CLTS*-Strategie abgeleitet wurde. Im *RP* ist zu erkennen, dass zwischen der *OV* `addBlock` und der *OV* `inp1`, die ursprüngliche *LV* mit dem Ende `+inport` in eine *NAC*-Kante umgewandelt und durch eine *LV* von kompatiblem Typ, mit den Enden `+block` und `+port`, ersetzt wurde.

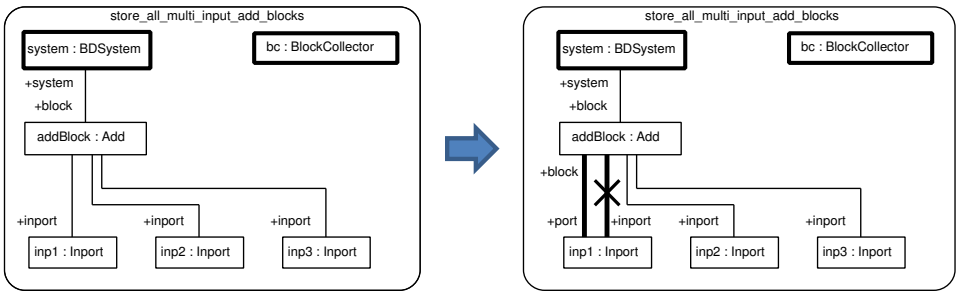


Abbildung 7.5: Zur Veranschaulichung der *CLTS*-Strategie am Beispiel

Die *CLTS*-Strategie umgeht das Problem von nicht durch einen Suchplan erreichbaren Zusammenhangskomponenten dadurch, dass mit der neuen *LV* eine Alternative zur ursprünglich möglichen Navigation geschaffen wird. Allerdings beinhaltet die Strategie eine andere Situation, die einer gesonderten Betrachtung bedarf. Weist nämlich eine der beiden beteiligten *OVs* bereits einen typgleichen Link im originären Muster auf und weist die zu dem *LV*-Typ zugehörige Assoziation am „abgewandten“ Ende – also dem Ende, das nicht an besagter *OV* endet – eine Multiplizitätsobergrenze von 1 auf, so würden vergeblich zwei entsprechende Links für ein Objekt gefordert. Dies würde zu Be-

[8] Man denke hierbei an Containment-Beziehungen oder Einschränkungen aufgrund der festgelegten Multiplizitäten.

dingungen führen, die niemals erfüllt werden können.[9] Deshalb müssen solche Kanten bei der Generierung der Modifikationen herausgefiltert und anschließend ignoriert werden. Grundsätzlich wirft dies auch die Frage auf, wie allgemein mit unerfüllbaren *RPs* umzugehen ist. Ein pragmatischer Ansatz bestünde darin, entsprechende *RPs* beim Testen zu identifizieren und bewusst von einer weiteren Betrachtung auszuklammern.

Erweitertes Anpassen des Link-Typs (CLTC - Change Link variable Type Complex). Beim Einsatz der zuvor beschriebenen *CLTS*-Strategie zeigt sich in der Praxis, dass es immer wieder vorkommen kann, dass kein alternativer *LV*-Typ zur Verfügung steht. Die möglichen Link-Typen werden durch das Metamodell definiert, wobei „allgemeinere" Beziehungen – beispielsweise Enthaltensein-Beziehungen – häufig auf Ebene von (abstrakten) Oberklassen beschrieben werden und „speziellere" Beziehungen – beispielsweise ein ausgezeichnetes spezielles (erstes, letztes oder aktuelles) Element – auf Ebene konkreter Unterklassen. So können zum Beispiel an den Link-Enden der betrachteten *LV* solche *OVs* verwendet werden, die Typen aufweisen, die Teil einer Vererbungshierarchie sind und relativ „weit oben" in dieser auftreten, die Klassen also einen relativ umfangreichen Vererbungsclan und eine relativ kleine Oberklassenmenge aufweisen. Wenn jetzt weitere Assoziationen teilweise erst für Unterklassen definiert sind und unter der Prämisse, dass i. d. R. Instanzen dieser Unterklassen im Modell auch tatsächlich auftreten, erscheint es sinnvoll, dass man eine *LV* auch durch eine Forderung nach einer *LV* mit „*teilkompatiblem*" Typ ersetzt. Damit dies möglich ist, muss neben der Konvertierung der alten *LV* zu einer *NAC* und der Einführung einer neuen *LV* vom Typ einer Assoziation, welche sich auf eine der Unterklassen einer der beiden ursprünglichen *OV*-Klassen bezieht, auch noch der Typ einer der beiden *OVs* zur benötigten Unterklasse abgeändert werden. Folglich entspricht die *CLTC*-Strategie der *CLTS*-Strategie, allerdings dass bei ersterer zusätzlich Änderungen des Typs bei einer der beteiligten *OVs* erlaubt sind, solange der neue Typ eine Unterklasse des alten Typs ist. Letzteres stellt sicher, dass die geänderte *OV* weiterhin valide ist. Die *OV*, deren Typ geändert werden soll, darf nicht als „`BOUND`" konfiguriert sein.

In Abbildung 7.6 ist ein entsprechendes Beispiel gegeben. Der Bereich links oben enthält den relevanten Ausschnitt des Blockdiagramm-Metamodells, vgl. auch Abbildung A.1. Ein Ausschnitt aus der zu testenden *SDM*-Spezifikation ist im oberen rechten Teil zu sehen. In der abgebildeten For-Each-Regel wird, ausgehend von einem gegebenen `Add`-Block, über alle `Port`-Instanzen iteriert, die über die Containment-Kante `hasPorts` (zwischen `Block` und der abstrakten Klasse `Port`) mit der gegebenen `Add`-Instanz verbunden sind. Will man nun die vorhandene *LV* des Musters, welche die `add`-*OV* mit der `port`-*OV* verknüpft, gemäß der *CLTS*-Strategie ersetzen, stellt man fest, dass keine alternativen Assoziationen im Metamodell vorhanden sind. Allerdings existieren die zwei Assoziationen `hasOutports` sowie `hasInports` zwischen der Klasse `Block` und den konkreten Unterklassen `Outport` und `Inport` der Klasse `Port`. Beide könnten nach einer Einschränkung des *OV*-Typs als Typ einer alternativen *LV* genutzt werden. Das Endergebnis der durch die *CLTC*-Strategie ableitbaren *RPs* ist im unteren Teil der Abbildung dargestellt: Es entstehen zwei *RPs*, in denen die ursprüngliche *LV* vom Typ

[9] Neben dem offensichtlichen Widerspruch, dass ein Objekt dann mehr als *die eine* erlaubte Beziehung des entsprechenden Typs eingehen würde, schließt die gewählte Abbildung auf die Code-Ebene das Erzeugen entsprechender Testmodelle bereits aus.

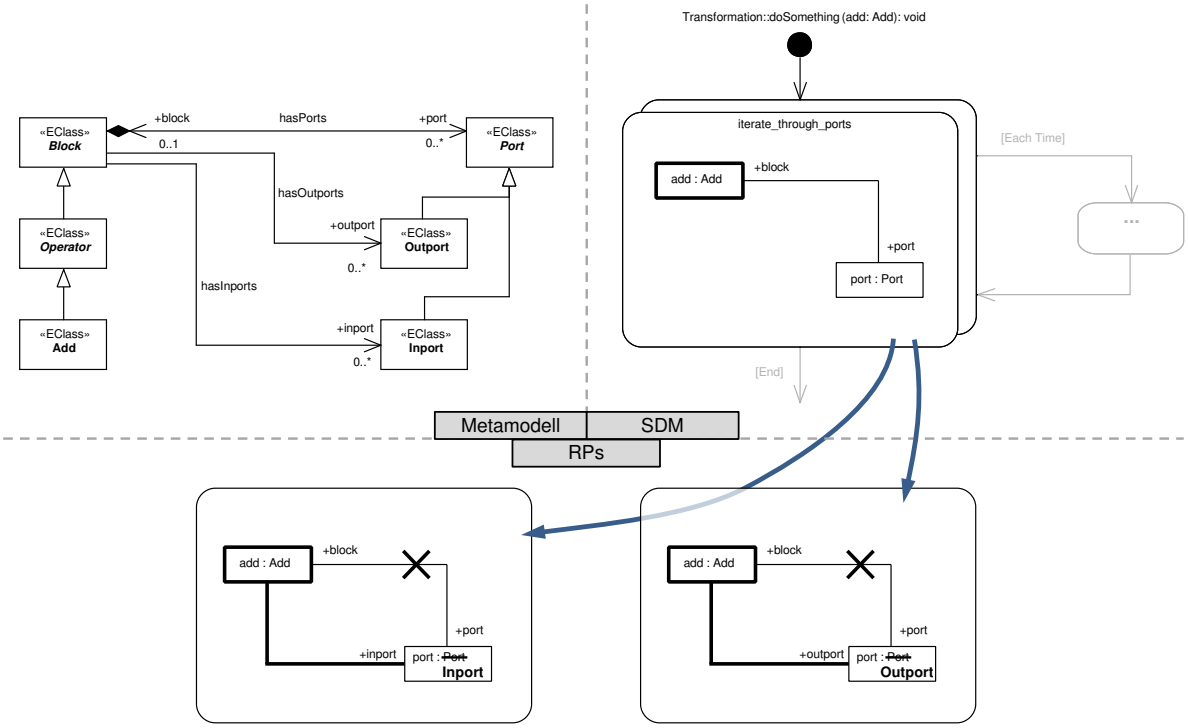


Abbildung 7.6: Zur Veranschaulichung der *CLTC*-Strategie am Beispiel (unter Vernachlässigung einer hier sowieso nicht vorhandenen Möglichkeit zur Definition von `subset`-Beziehungen zwischen Link-Enden)

`hasPorts` einmal durch eine *LV* vom Typ `hasOutports` und einmal durch eine vom Typ `hasInports` ersetzt wurde. Entsprechend wurde der Typ der `port`-*OV* jeweils angepasst.

Zulassen von nicht-isomorphem Binden (ANIB - Allow Non-Isomorphic Binding) Ein häufiger anzutreffender Fehler bei der Entwicklung mit der *SDM*-Sprache und wahrscheinlich ebenso beim Einsatz anderer Graphtransformationssprachen besteht darin, dass sehr viele Fälle übersehen werden, in denen als separate Knoten modellierte Musterteile eigentlich auch auf identische Objekte im Instanzgraphen abgebildet werden könnten bzw. sogar müssten. Die Semantik von *SDM*-Regeln ist, wie in Kapitel 4.3 beschrieben, dass nur *injektive* Treffer des jeweiligen Musters gesucht werden. Die einzelnen *OVs* von gleichem oder ähnlichem Typ dürfen folglich nur auf unterschiedliche Instanzen im Modell abgebildet werden. Für ein hinreichend gründliches Testen wäre es dennoch sinnvoll, auch Fälle abzudecken, in welchen die Testmenge nichtisomorphe Treffer enthält, so denn diese möglich wären. Hierdurch könnte überprüft werden, ob sich die Transformation auch für solche Strukturen erwartungsgemäß verhält und somit die sog. *Injektivitätschecks* korrekt funktionieren. Letztere lassen sich in der hier verwendeten *SDM*-Implementierung zwar nicht explizit deaktivieren, könnten bei anderen Sprachen bzw. Sprachvarianten aber durchaus auch fälschlicherweise ausgeschlossen werden. Darüber hinaus ist zu erwarten, dass eine durch entsprechende *RPs* bedingte Auseinandersetzung mit diesem speziellen Teilaspekt der Spezifikation den positiven Nebeneffekt hat, dass sich ein Entwickler eventuell doch übersehenen Anwendungssituationen bewusst wird.

In seiner Habilitationsschrift beschreibt Zündorf ab Seite 147 ff. beispielsweise das *May-*

be-Konstrukt für seine ursprüngliche Variante der Story-Diagramm-Sprache, vgl. [Zun02]. Durch dieses Konstrukt ist es dem Entwickler möglich, eine nicht-injektive Abbildungen von Teilen des Musters in den Objektgraphen explizit zuzulassen. Hierzu werden einige der *typkompatiblen OVs* eines Musters durch einen entsprechenden *Maybe*-Ausdruck miteinander verknüpft, wodurch zugelassen wird, dass einige oder alle verknüpfte *OVs* an das selbe Objekt aus dem Modell gebunden werden können.

Die hier nur skizzierte *ANIB*-Strategie setzt ein solches *Maybe*-Konstrukt voraus, was hier leider nicht gegeben ist. Durch die Strategie wird gefordert, dass für alle[10] möglichen Teilmengen der Typ-kompatiblen *OVs* eines Musters jeweils ein *RP* abgeleitet wird, das ein *Maybe*-Konstrukt enthält, welches die Elemente der gerade betrachteten Teilmenge miteinander verknüpft. Zwar könnte das *Maybe*-Konstrukt mit Hilfe mehrerer Regeln und des Kontrollflusses simuliert werden, was darauf hindeutet, warum sein Fehlen in der Praxis kein echtes Problem darstellt. Diese Art der Implementierung hätte für den hier betrachteten Anwendungsfall allerdings den gravierenden Nachteil, dass ein entsprechendes Coverage-Item nicht mehr in Form eines einzelnen einfachen Musters formuliert werden könnte. Das *Maybe*-Konstrukt erscheint über das Testen hinaus wünschenswert. Solange es aber noch nicht verfügbar ist,[11] kann auch die *ANIB*-Strategie nicht einfach umgesetzt werden.

Verschmelzen von Object-Variablen (MV - Merge OVs) Da die *ANIB*-Strategie einerseits aufgrund der erwähnten praktischen Einschränkungen nicht umgesetzt werden kann, andererseits durch sie lediglich *erlaubt*, aber nicht *gefordert* wird, dass *OVs* an identische Objekte gebunden werden, sind hierfür alternative Ansätze gefragt. Die im Rahmen der Arbeit angedachte *MV*-Strategie ist ein entsprechender Entwurf für eine solche Alternative. Im Rahmen dieser Strategie wird versucht, mindestens zwei Typ-kompatible *OVs* innerhalb eines Musters zu einer einzigen *OV* zu verschmelzen. Die resultierenden *RPs* fordern dann Testmodellen ein, in denen die Bilder mehrerer ursprünglich separater *OVs* auf ein gemeinsames Objekt im Modell fallen.

Praktisch umgesetzt werden kann die Strategie, indem die zu verschmelzenden *OVs* aus dem Muster entfernt und durch eine neue *OV* mit passendem Typ ersetzt werden. Kanten, die ursprünglich an einer der betrachteten Variablen enden oder entspringen, werden dabei auf den neu entstandenen Knoten umgebogen. Offen bleibt, wie mit unerfüllbaren resultierenden Forderungen, z. B. nach mehreren Links desselben Typs zwischen den gleichen *OVs*, umgegangen werden sollte. Problematisch könnten auch Multiplizitätsgrenzen im Metamodell sein, die verhindern würden, dass die eingeforderte Menge an Links existieren kann. Möglich wäre das Entfernen entsprechender Kanten oder aber das Ausschließen der *OV*-Verschmelzung für solch problematischen Fälle. Als *kompatible* können solche *OVs* gelten, bei denen die Schnittmenge der erweiterten Vererbungsclans ihrer Typen nicht leer ist. So sind beispielsweise Klassen A und B genau dann *kompatibel*, wenn gilt $\exists c \in \text{clan}^*(A) : c \in \text{clan}^*(B)$. Als Typ der verschmolzenen *OV* muss eine Klasse aus der Schnittmenge $\text{clan}^*(A) \cap \text{clan}^*(B)$ gewählt werden, die – als zusätzliche Nebenbedingung, zur Vermeidung von zu starken Einschränkungen – noch möglichst weit

[10] Bei Bedarf ließe sich dies auf einige repräsentative Teilmengen eingrenzen, etwa mit Optimierungsalgorithmen, die das Mengenüberdeckungsproblem sinnvoll (näherungsweise) lösen.

[11] Eine große Herausforderung, bezogen auf die Einführung eines solchen Konstrukts, liegt beispielsweise in der Festlegung einer Bedeutung von *Maybe*-Ausdrücken für im Kontrollfluss nachfolgende Regeln sowie den Bindungsstatus einbezogener Variablen.

„oben“ in der Vererbungshierarchie auftreten sollte. Dies trifft insbesondere auf solche Klassen in $\text{clan}^*(A) \cap \text{clan}^*(B)$ zu, deren jeweilige Oberklassenmenge keine Elemente aus der Schnittmenge der erweiterten Vererbungsclans enthält.

Aus praktischen Überlegungen heraus ist diese Strategie allerdings mehrfach als problematisch anzusehen. So ist einerseits offen, wie mit bereits gebundenen *OVs* im Rahmen einer solchen Strategie verfahren werden sollte. Man könnte sich vorstellen, diese ebenfalls mit anderen (ungebundenen oder gebundenen) *OVs* zu verschmelzen, was auf jeden Fall zu einer ebenfalls gebundenen verschmolzenen Variable führen muss, da im Anschluss an das Verschmelzen ansonsten ggf. zu wenige Kontextobjekte existieren würden. Mehrere gebundene *OVs* zu verschmelzen führt im Allgemeinen zu Einschränkungen möglicher legaler Belegungen, die bereits beim eigentlichen Bindevorgang hätten Berücksichtigung finden müssen. Dies würde allerdings deutliche Anpassungen an der zu testenden Transformation respektive der aufrufenden Stelle dieser verlangen, was hier nicht gewünscht ist. Auch das Verschmelzen einer gebundenen mit einer ungebundenen *OV* ist problematisch, da sich durch die Verschmelzung nur in sehr seltenen Ausnahmefällen[12] keinerlei zusätzliche Einschränkungen für den verschmolzenen Knoten im Vergleich zur gebundenen *OV* ergeben würden. Zusätzliche Einschränkungen im Nachhinein sind, wie bereits festgestellt, aber problematisch. Somit ist die Einbeziehung von gebundenen Variablen im Rahmen der Strategie zumindest unpraktikable.

Ein weitere Punkt ist, dass auch Links zwischen den zu verschmelzenden Knoten problematisch sein können – unmittelbar offensichtlich wird dies, wenn es sich dabei um Containment-Kanten handelt. Das Verschmelzen würde dann in einer Selbstreferenz resultieren, durch die die unerfüllbare Forderung aufgestellt werden würde, dass ein Objekt sein eigener Container sein muss. Auch ist das Verschmelzen im Hinblick auf Attributbedingungen alles andere als trivial, so dass diese entweder ebenfalls ausgeklammert oder aufwendig analysiert werden müssten. Aufgrund der vielen offenen Fragen zu einer Realisierung, die im Rahmen dieser Arbeit aus Zeitgründen nicht abschließend geklärt werden konnten, wurde eine Umsetzung dieser Strategie nicht weiter verfolgt.

Übersicht der Strategien In Tabelle 7.1 ist zum Abschluss noch eine Übersicht der bisher vorgestellten *RP*-Generierungsstrategien gegeben. Neben den zuvor verwendeten Kürzeln enthält die Tabelle Angaben dazu (i) wo die Strategie im Muster ansetzt bzw. welche Elemente sie verändert, (ii) wie die vorzunehmenden Änderungen am Ausgangsmuster hin zum *RP* zu charakterisieren sind, (iii) wie es um die Umsetzung der Strategie, bezogen auf die im folgenden genutzte *RP*-Implementierung gestellt ist, (iv) wie aufwendig die Umsetzung der Strategie, im Hinblick auf eine Implementierung anzusehen ist und (v) wie viele *RPs* durch die Anwendung der Strategie auf eine Regel entstehen.

7.2.4 Zur Instrumentierung

Der Instrumentierungsvorgang der ursprünglichen *SDM*-Transformation, welcher die technische Grundlage zur Ermittlung der *RP*-Überdeckung während der Testausführung repräsentiert, wurde bereits mehrfach erwähnt, s. Teilschritte ⑥ und ⑦ in Abbildung 7.2

[12] Ein solcher Fall liegt beispielsweise dann vor, wenn (a) Attributbedingungen nicht vorliegen, (b) die zu verschmelzenden *OVs* nicht untereinander verbunden sind, also keine Selbstreferenzen entstehen würden, und (c) ansonsten alle Links der ungebundenen Variable beim Verschmelzen zu redundanten, auszufilternden Forderungen führen würden.

Nr.	ID	Ändert...	Änderungsart	Umgesetzt?	Aufwand	#RPs
1	UC	—	aufweichend	✓	gering	1
2	RAN	NACs	—	✓	gering - mittel	1
3	REMAN	NACs	aufweichend	✗	gering - mittel	1
4	COT	OVs	je nachdem	✓	aufwendig	n
5	ROOV	OVs	einschränken	✓	gering - mittel	n
6	EOOV	OVs	einschränken	✗	gering - mittel	n
7	EMOV	OVs	—	✓	mittel - aufwendig	n
8	EMLV	LVs	—	✗	mittel - aufwendig	n
9	ROLV	LVs	einschränken	✓	gering	n
10	EOLV	LVs	einschränken	✗	gering	n
11	CLTS	LVs	je nachdem	✓	mittel - aufwendig	n
12	CLTC	LVs + OVs	je nachdem	✓	aufwendig	n
13	ANIB	OV-Binding	aufweichen	✗	mittel	n
14	MV	OVs (+ LVs)	—	✗	sehr aufwendig	n

Tabelle 7.1: Übersichtstabell zu den RP-Generierungsstrategien

sowie die Zeile 17 in 1. Solange eine Regel in einem einfachen Story-Knoten ohne For-Each-Semantik enthalten ist und dieser auch nur eine einzige eingehende Kontrollflusskante besitzt, ist der Instrumentierungsvorgang zur Auswertung der zugehörigen *RPs* sehr einfach: (i) ein neuer Kontrollflussknoten, der den Aufruf der RP-Auswertung kapselt, wird erzeugt, (ii) die Kontrollflusskante, die vom ursprünglichen Vorgängerknoten auf den Knoten der betrachteten Regel zeigt, wird auf diesen neuen Knoten umgebogen und (iii) ausgehend von dem neuen Knoten wird eine neue Kontrollflusskante zum Knoten der betrachteten Regel erzeugt.

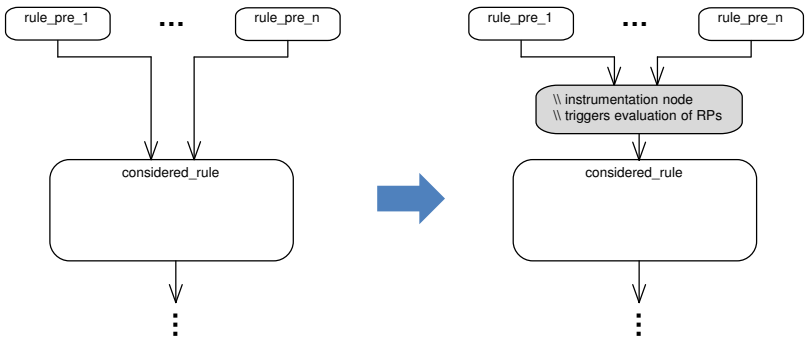


Abbildung 7.7: Instrumentierung bei Story-Pattern mit mehreren eingehenden Kanten

Weist der *SDM*-Knoten der betrachteten Regel allerdings mehrere eingehende Kontrollflusskanten auf, so muss sichergestellt werden, dass die Instrumentierung auf jedem der möglichen Abläufe zum Tragen kommt. Eine solche Ausgangssituation ist im linken Bereich der Abbildung 7.7 skizziert. Die Struktur nach der Instrumentierung ist rechts daneben dargestellt. Es wird genau ein neuer Instrumentierungsknoten in den Kontrollflussgraphen eingebaut. Die Alternative besteht darin, einen solchen Knoten für jede der n eingehende Kanten einzubauen. Dies würde allerdings zu unnötiger Redundanz führen.

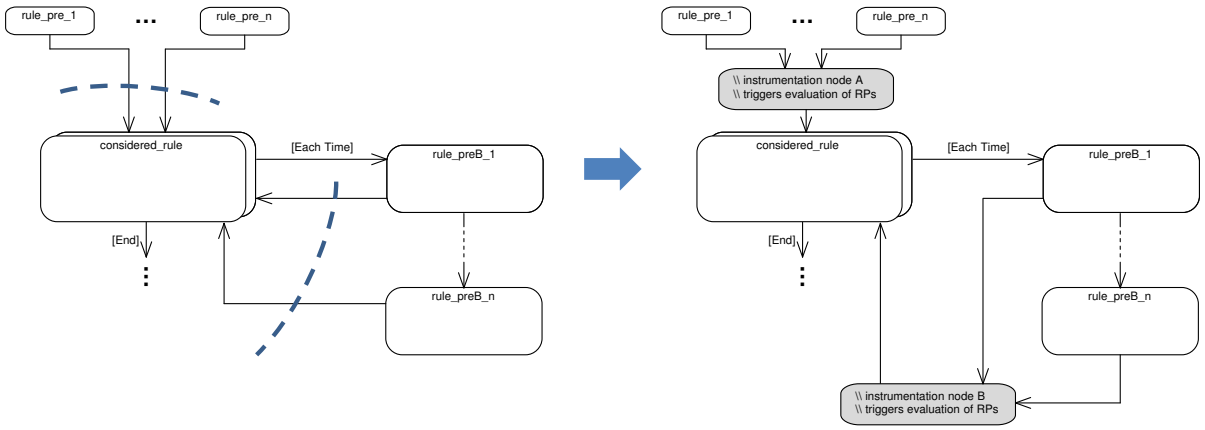


Abbildung 7.8: Instrumentierung bei Story-Pattern mit For-Each-Semantik

Etwas anders stellt sich die Situation dar, wenn der Story-Knoten der betrachteten Regel als Each-Time-Knoten konfiguriert wurde. Falls der For-Each-Knoten eine *ausgehende* Each-Time-Kante aufweist, ist die Menge der eingehenden Kanten des Knotens, im Folgenden mit I bezeichnet, in zwei Partitionen aufgeteilt, nämlich in „echte“ eingehende Kanten I' und „zurückführende“ Kanten I_r aus der Each-Time-Komponente. Formaler ausgedrückt enthält I' alle eingehenden Kanten, für die gilt, dass sie Teil eines Pfades entlang der entgegengesetzten Kontrollflusskantenrichtung sind, der bei dem betrachteten For-Each-Knoten beginnt und zum einzigen Start-Knoten führt, ohne dass dabei der betrachtete For-Each-Knoten erneut besucht wird. I_r sind alle übrigen eingehenden Kanten des betrachteten Knotens ($I_r = I \setminus I'$).

Für die Menge I' muss genau ein Instrumentierungskonten erstellt werden und die Kanten aus I' entsprechend auf diesen umgebogen werden, vgl. den oberen grau eingefärbten Knoten im rechten Teil von Abbildung 7.8. Ob für die Menge I_r auch ein Instrumentierungskonten erzeugt werden sollte, hängt von der Umsetzung des Each-Time-Verhaltens ab. Erfolgt eine erneute Auswertung des Musters der Regel nach jedem Durchlauf der Each-Time-Komponente, so sollte ein zweiter Instrumentierungsknoten erstellt werden. Dieser Fall ist ebenfalls in Abbildung 7.8 gezeigt, vgl. den unteren grau hinterlegten Knoten. Ist das Each-Time-Verhalten allerdings strikt umgesetzt, so dass die Auswertung der Each-Time-Komponente keinen Einfluss mehr auf die Auswertung des Musters im Each-Time-Knoten haben kann, so ist der zweite Knoten überflüssig und kann entfallen.

Gesetzt den Fall, dass eine Each-Time-Kante, wie in Abbildung 7.8 im linken Bereich dargestellt, vorliegt, muss die Instrumentierung dergestalt erfolgen, dass zwei separate Instrumentierungsknoten in den Kontrollfluss eingebaut werden. Würde man nur einen Instrumentierungsknoten verwenden, so würde dieser gleichzeitig in zwei Gültigkeitsbereichen bzw. Scopes existieren, was bei *SDM*-Transformationen per Definition ausgeschlossen wird. Für eine Realiserung werden alle eingehenden Kanten der Menge I' auf eine Kopie des Instrumentierungsknotens umgebogen, und alle Kanten der Menge I_r auf die zweite Kopie des Instrumentierungsknotens. Diese Variante ist im rechten Teil der Abbildung 7.8 dargestellt.

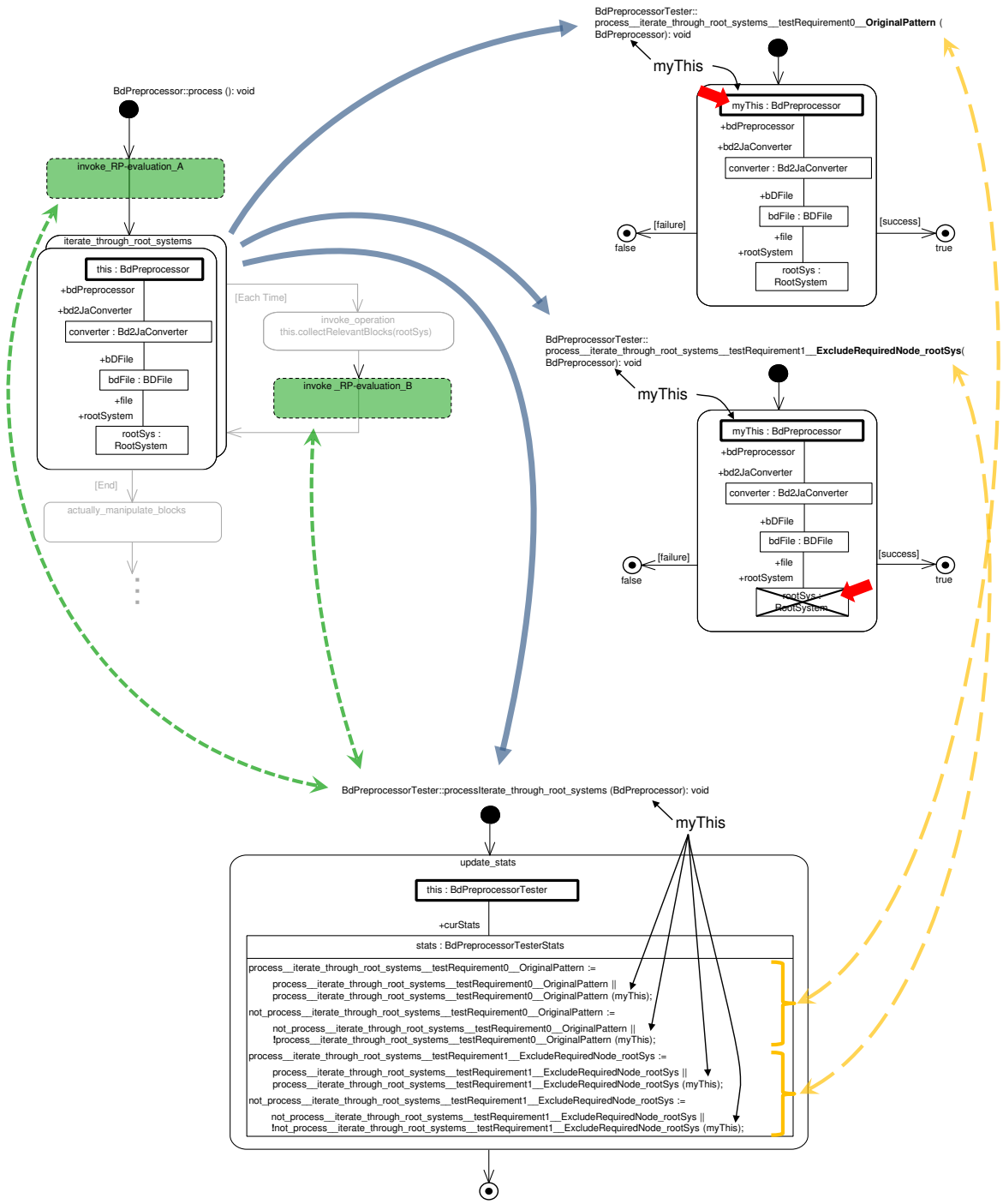


Abbildung 7.9: Instrumentierung am konkreten Beispiel

In Abbildung 7.9 ist die Ableitung von zwei konkreten *RPs* aus einem Muster der Beispieltransformation gezeigt. Auch der entsprechende Auswertungscode mit dem Story-Knoten „update_stats“ ist für das konkrete Beispiel gezeigt. Er sorgt auch für die Aufrufe der *RP*-Auswertungsoperationen. Die beiden Instrumentierungskonten in der „process“-Operation sorgen dafür, dass der Auswertungscode zu dem richtigen Zeitpunkten aufgerufen wird.

7.3 Hinweise zur Implementierung

Nach der Beschreibung des *RPC*-Ansatzes auf konzeptioneller Ebene werden nun einige Details der realisierten technischen Umsetzung erläutert. Im Rahmen der Arbeit entstand als Umsetzung des *RPC*-Ansatzes eine Erweiterung der eMoflon-Tool-Suite in Form eines Plugins für die Eclipse-*IDE*. Das Hauptziel der Entwicklung lag in einer möglichst großen Automatisierung wiederkehrender Teilschritte, die ein Entwickler beim Einsatz des *RPC*-Ansatzes durchführen muss. Die entsprechenden Funktionalitäten sollten sich dabei möglichst nahtlos in die bestehende Werkzeuglandschaft integrieren.

Die entstandene Implementierung umfasst zwei Kernfunktionalitäten. Einerseits die erwähnten Erweiterungen der Eclipse-*IDE* und andererseits Funktionen zur initialen Generierung eines problemspezifischen Testaufbaus zur Ausführung und Analyse der Tests auf Basis des weit verbreiteten JUnit4-Rahmenwerks. Entsprechend dieser Zweiteilung ist auch dieses Unterkapitel in zweit Abschnitte unterteilt.

7.3.1 Eclipse

An der Erweiterung von Eclipse und eMoflon sind insgesamt vier Entwicklungsprojekte beteiligt: (i) ein Metamodell-Projekt (RpCoverageMetamodel) für eMoflon, (ii) ein daraus abgeleitetes eMoflon-Repository-Projekt[13] (RpCoverageMetamodelSpec), welches zusätzlich auch als Eclipse-Plugin-Projekt konfiguriert ist, (iii) ein Hilfsplugin zur Visualisierung[14] der generierten *RPs* (SdmDiagramHandler) sowie (iv) ein Projekt, welches das Haupt-Plugin SdmTestCoverageFramework definiert und das im Kern die eigentlichen Eclipse-Anpassungen sowie die *RPC*-Funktionalität umsetzt. Letzteres nutzt das in (i) definierte Metamodell indirekt, indem es den daraus generierten Code aus (ii) einbindet.

Das `SdmTestCoverage`-Plugin

Im Folgenden geht es um die Beschreibung des Haupt-Plugins. Dessen wichtigster Anwendungsfall besteht darin, dass ein Benutzer ein Eclipse-Projekt, welches die zu testenden Transformation definiert, so anpasst, dass anschließend eine Analyse der Testüberdeckung beim Ausführen der Tests möglich ist.

In Abbildung 7.10 sind die wesentlichen Schritte in Form eines Aktitvitätsdiagramms aufgeführt, die notwendig sind, um für ein eMoflon-Repository-Projekt die Bestimmung

[13] Bei diesem Projekttyp handelt es sich um ein gewöhnliches Java- bzw. Plug-in-Projekt aus dem Standardrepertoire von Eclipse, erweitert um einen sogenannten *Builder* für die eMoflon-Codegenerierung.

[14] In der Implementierung erfolgt die Visualisierung mit Hilfe des *Fujaba4Eclipse*-Plugins, v0.8.1, der Software Engineering Group der Universität Paderborn, s. `http://dsd-serv.uni-paderborn.de/svn/updatesites/trunk/de.uni_paderborn.fujaba4eclipse.updatesite` (zuletzt am 28.3.2014 erfolgreich abgerufen)

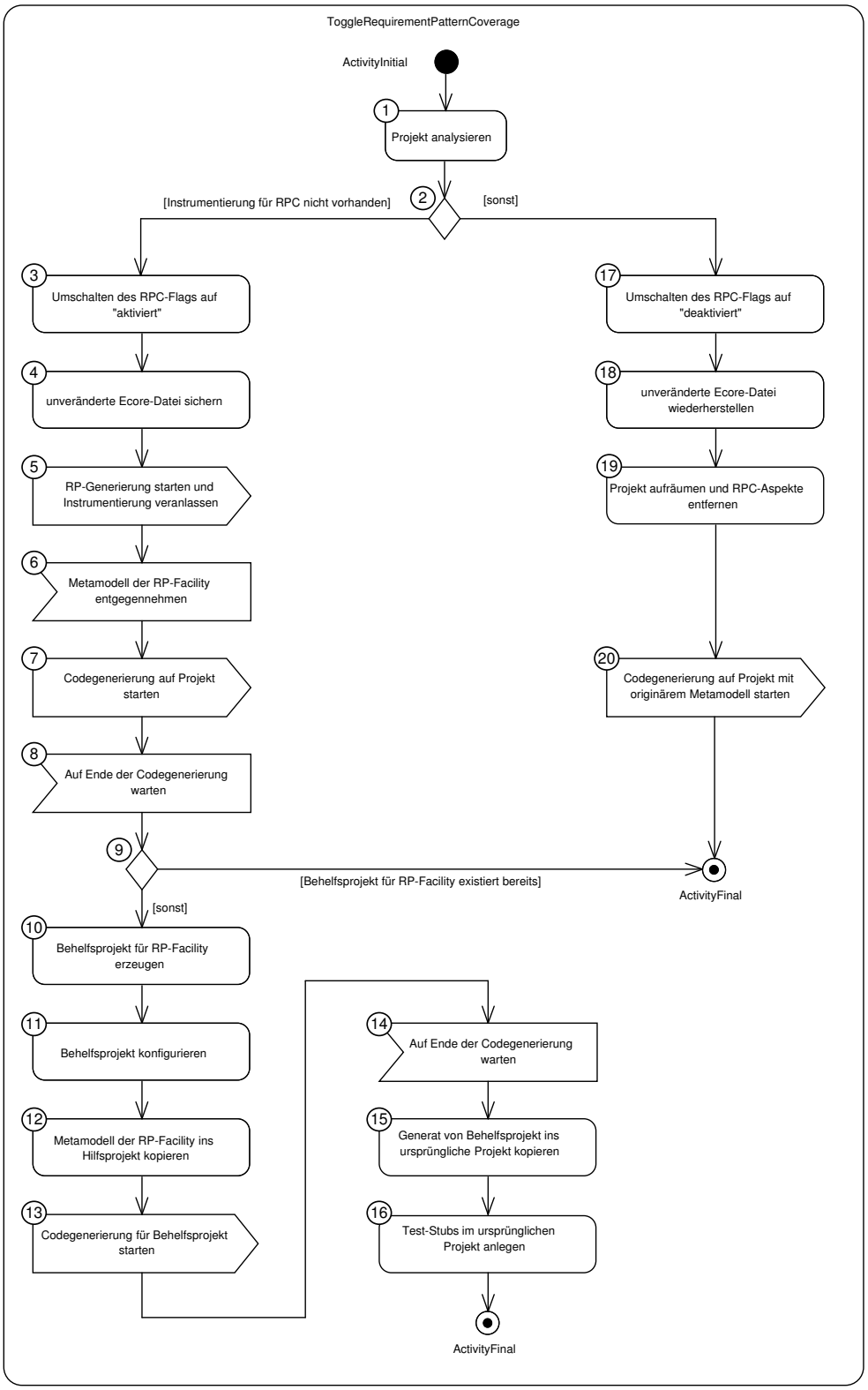


Abbildung 7.10: Ein- und Ausschalten der Erfassung der *RPC*-Werte für ein Transformationsprojekt

der Überdeckung einzuschalten bzw. wieder rückgängig zu machen. Letzteres ist wünschenswert, da das ursprüngliche Modell bzw. die ursprüngliche Transformationsbeschreibung durch den Instrumentierungsschritt verändert wird. Diese Änderungen fließen nicht in die der ursprünglichen Transformation zugrunde liegende *SDM*-Beschreibung zurück, so dass sich die Variante ohne Instrumentierung folglich aus dieser *SDM*-Spezifikation restaurieren lässt. Um negative Effekte aufgrund der Instrumentierung, wie beispielsweise längere Ausführungszeiten, zu umgehen, erscheint es dennoch sinnvoll, automatisiert zur ursprünglichen Variante zurückkehren zu können. Damit die Änderungen durch die Instrumentierung direkt erkennbar sind, wird dem Projekt eine entsprechende Zustandsvariable – in der Darstellung als *RPC-Flag* bezeichnet – gegeben.

Die Abbildung 7.10 zeigt beide möglichen Abläufe, in Abhängigkeit davon, ob das Projekt bisher noch nicht für die Bestimmung der *RPC*-Werte konfiguriert ist (Teilschritte 1 bis 16) oder eben doch (Teilschritte 1, 2, 17, . . . , 20). In den Schritten ① und ② wird das Projekt zuerst analysiert und anschließend entsprechend der beiden Optionen verzweigt. In dem Fall, dass die *RPC*-Werte für das Projekt noch nicht bestimmt werden, wir anschließend in Schritt ③ das *RPC-Flag* auf `true` (also „aktiviert“) gesetzt. Danach wird in Schritt ④ die Serialisierung der ursprünglichen Transformationsbeschreibung samt Metamodell, in Form einer sog. .ecore-Datei, gesichert. Sie enthält neben der Definition der Klassen und Assoziationen auch die Serialisierungen der ursprünglichen *GTs*, die getestet werden sollen. Durch das Sichern dieser Datei lässt sich das ursprüngliche *SUTs* später wieder herstellen und man kann in Teilschritt ⑤ gefahrlos die Ableitung der *RPs* inklusive des Instrumentierungsschritts starten. Die eigentliche Ableitung der *RPs* erfolgt im wesentlichen so, wie bereits beschrieben. Die Aktion ⑤ stellt einen Aufruf eines komplexen Arbeitsschritts dar, weshalb sie hier als Versenden einer Nachricht modelliert wurde, um damit anzudeuten, dass dieser Teil extern geschieht. Im Rahmen des externen Arbeitsschritts wird die ursprüngliche .ecore-Datei aufgrund der Instrumentierung der Transformationsimplementierungen verändert. In Teilschritt ⑥ wird auf das Endergebnis der *RP*-Generierung gewartet und die Metamodellrepräsentation der *RP-Facility* wird in Form einer zweiten .ecore-Datei entgegen genommen, welche im weiteren Verlauf noch Verwendung findet. Im Anschluss daran ist sichergestellt, dass die Instrumentierung der originären *SDMs* abgeschlossen ist, und so kann in Teilschritt ⑦ der Codegenerator auf dem Repository-Projekt gestartet werden, was dazu führt, dass für die instrumentierten *SDM*-Varianten Code erzeugt wird. In ⑧ wird das Ende dieses Schrittes abgewartet, bevor dann in Schritt ⑨ entschieden wird, ob das benötigte temporäre Behelfsprojekt zur Codegenerierung aus dem abgeleiteten *RP-Facility*-Metamodell wirklich erzeugt werden kann. Ist dies der Fall, wird mit Schritt ⑩ fortgefahren, in dem das Behelfsprojekt zuerst erzeugt wird, bevor es anschließend in ⑪ für die Codegenerierung konfiguriert wird. Bei negativer Bewertung in Schritt ⑨ endet die Ausführung. Die Konfiguration aus Schritt ⑪ beinhaltet einerseits das Setzen sog. *Eclipse-Natures*, also das Festlegen der Art des Projektes und die Bestimmung der standardmäßig benötigten *Builder* für das Projekt. Andererseits umfasst es die Definition der für die Codegenerierung und das Kompilieren der generierten Java-Dateien benötigten Projekte, von denen das Behelfsprojekt selbst abhängt. Erst danach stellt das Behelfsprojekt ein vollständiges, korrekt konfiguriertes eMoflon-Repository-Projekt dar, mit dessen Hilfe aus dem *RP-Facility*-Metamodell Code erzeugt und anschließend kompiliert werden kann. Um letzteres tatsächlich tun zu können, muss zuvor aber noch in Teilschritt ⑫ die .ecore-Datei des *RP-Facility*-Metamodells als Eingabe in das Behelfsprojekt kopiert werden, wonach dann in ⑬ die Codegenerierung

gestartet und das Ergebnis in Schritt ⑭ erwartet wird. Nachdem der Code vollständig generiert ist, wird das Generat des Behelfsprojektes in Teilschritt ⑮ in das ursprüngliche Projekt kopiert. Dies ist notwendig, da es durch eine ansonsten benötigte Projektabhängigkeit zu einer zyklischen Abhängigkeit zwischen dem ursprünglichem und dem Behelfsprojekt kommen würde. Das Kopieren des Codes umgeht die Abhängigkeit, indem beide Projekte de facto verschmolzen werden. Als letzte Aktion des Ablaufs werden in Teilschritt ⑯ noch Test-Rümpfe im ursprünglichen Projekt generiert. Auf diesen letzten Punkt wird in den nachfolgenden Abschnitten noch gesondert eingegangen.

Visualisierung und Coverage-Auswertung

Durch das Rahmenwerk werden beim Aktivieren der *RPC*-Erfassung für ein Projekt auch Test-Rümpfe generiert. Welche Artefakte dabei konkret erzeugt werden, wird in Abschnitt 7.3.2 ausgeführt. Hier sei festgehalten, dass diese generierten Teile zur Erfassung und Visualisierung der Testüberdeckung von zentraler Bedeutung sind. Einerseits sorgen sie dafür, dass die Werte der *RP*-Überdeckung beim Testen konsequent und über mehrere Läufe hinweg stabil, d. h. mit stets fester Ausführungsreihenfolge der Tests erfasst werden. Andererseits wird durch sie sichergestellt, dass Überdeckungsdaten an die im folgenden beschrieben Visualisierungskomponente des `SdmTestCoverage`-Plugins übermittelt werden.

Eine kompakte, visuelle Aufbereitung der Überdeckung erscheint sinnvoll, um sich schnell einen groben Überblick verschaffen zu können und nicht ausreichend getestete Teile der Transformation unmittelbar zu erkennen. Im oberen der zwei in Abbildung 7.11 gezeigten Eclipse-Fenster ist die realisierte Visualisierungskomponente des Eclipse-Plugins zu sehen. Die Abbildung zeigt einen Ausschnitt der gemessenen Überdeckung für die `convert`-Operation der Klasse `Bd2JaConverter` des Beispiels (für Details s. Abbildungen A.3, A.9 und A.10). Die Darstellung der *RPC*-Werte erfolgt mittels hierarchischer Baumdarstellung, die selbst wiederum Teil einer Tabelle ist. Unterhalb eines Projektbezeichners werden zuerst die Metamodellklassen und darunter die vorhandenen Operationen gruppiert. Für Operationen wird im rechten Bereich der Tabelle in der Spalte *Coverage* eine zweifarbige Darstellung generiert, die das Verhältnis von überdeckten (grüner Teilbalken) zu nicht überdeckten (roter Teilbalken) Coverage-Items widerspiegelt. Unterhalb der Operationen enthält der Baum Einträge für die einzelnen Coverage-Items. Die Darstellung in der *Coverage*-Spalte zeigt an, ob das entsprechende Item im Rahmen der Testausführung überdeckt (grüner Kreis) oder nicht überdeckt (roter Kreis) wurde. Zusätzlich enthält jede Zeile für ein Coverage-Item einen *Index* (Spalte „First Occurrence Index“), der angibt, ab welchem Test bzw. ab welchem *Tick* – normalerweise wird dieser am Ende eines jeden Tests ausgelöst – das Item als überdeckt gilt. Ein Index von -1 steht für „nicht überdeckt“.

Im unteren Bereich der Abbildung 7.11 ist die Standard-Visualisierung von JUnit in Eclipse für das gleiche Beispiel gezeigt. Da die generierten Test-Rümpfe auf dem JUnit-Rahmenwerk aufbauen, lässt sich die entsprechende Maschinerie inkl. der Anzeige wiederverwenden. Durch einen generierten JUnit-Test-Runner wird sichergestellt, dass die Reihenfolge der JUnit-Tests von Lauf zu Lauf stabil bleibt, so dass die Indices der *RPC*-Visualisierung zu dieser Reihenfolge passen. Aus Benutzersicht ist darüber hinaus die Funktion, mit welcher man sich für ein bestimmtes Coverage-Item eine grafische Darstellung der generierten *RP*-Auswertungsoperation ausgeben lassen kann, sehr hilfreich.

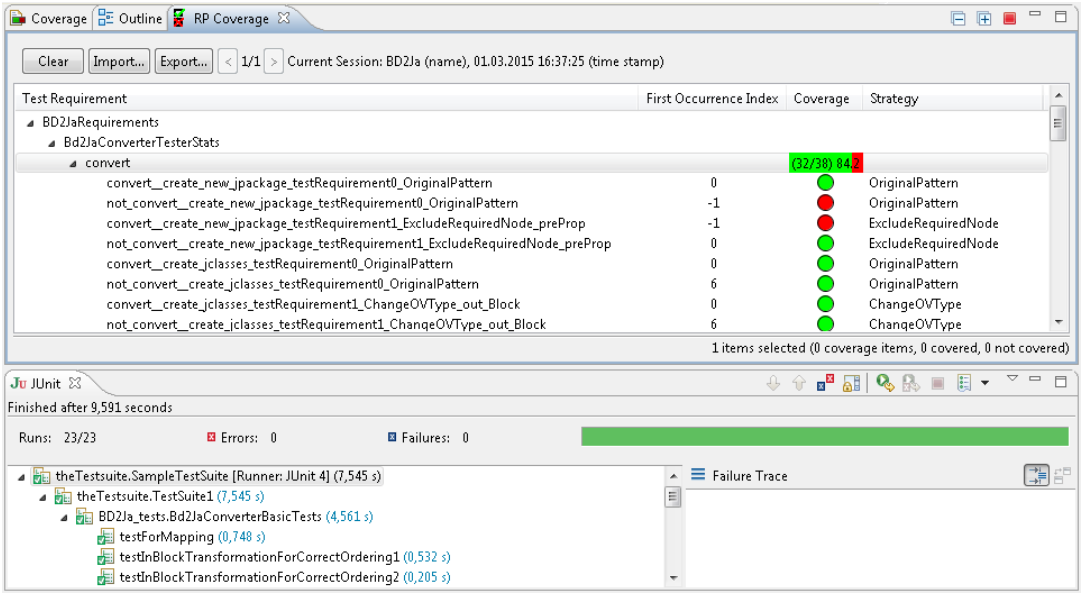


Abbildung 7.11: Visualisierung der *RP*-Überdeckung - Eclipse-View mit den *RPC*-Werten der Beispieltransformation (darunter die normale JUnit-View)

Per Doppelklick auf ein Coverage-Item öffnet sich eine entsprechende Darstellung des zugehörigen *RP*, welche technisch auf *Fujaba4Eclipse*[14] aufbaut.

Abbildung 7.12 zeigt das Metamodell, welches dem Datenmodell der Visualisierung entspricht. Einerseits werden mit Hilfe entsprechender Instanzen die Überdeckungsdaten des aktuellen und vergangener Testläufe verwaltet, andererseits bietet das EMF-basierte Metamodell eine einfache Möglichkeit zur Serialisierung der gewonnene Messwerte, um so eine spätere Nachvollziehbarkeit und Dokumentation zu ermöglichen. Die Instanzen des Metamodells werden zur Laufzeit von der Testumgebung angelegt und befüllt, indem Nachrichten und Signale, die von der eigentlichen Laufzeitumgebung der Testausführung aus an die Eclipse-Visualisierung gesendet werden, in entsprechende Aktionen und Modifikationen übersetzt werden. Ein einfaches Kommunikationsprotokoll sorgt dafür, dass Signale mehrerer sequenziell aufeinanderfolgender Durchläufe unterschieden und verwaltet werden können.

Wurzelelement der Enthaltenseinshierarchie im Metamodell ist die `RpCoverageModel`-Klasse. Sie beinhaltet mehrere `Session`-Objekte, die sie über das `session`-Ende der Containment-Kante verwaltet. Außerdem zeigt die Referenz `currentSession` auf die aktuell angezeigte `Session`-Instanz. Eine `Session` besitzt einen Zeitstempel (`timestamp`) und speichert den Namen des Projektes, für das die Überdeckung ermittelt wird. Außerdem dient sie als Container für `TestSuite`-Instanzen, welche wiederum `Test`-Instanzen gruppieren, sowie für sog. `RpCoverageItem`-Objekte. `TestSuite`- und `Test`-Instanzen beziehen sich hierbei auf die namensgleichen JUnit-Konzepte und geben so einen Hinweis darauf, welche Tests ausgeführt wurden. Die Instanzen der konkreten `RpCoverageItem`-Unterklassen repräsentieren die eigentlichen Überdeckungsdaten, die in der Benutzeroberfläche, im Folgenden als *Graphical User Interface (GUI)* bezeichnet, dargestellt werden. Dabei bilden die `RpCoverageItem`-Klassen die hierarchische Struktur des *SUT*-Metamodells in Form von Paketen (`RpContainerForPackage`), Klassen (`RpContainer-`

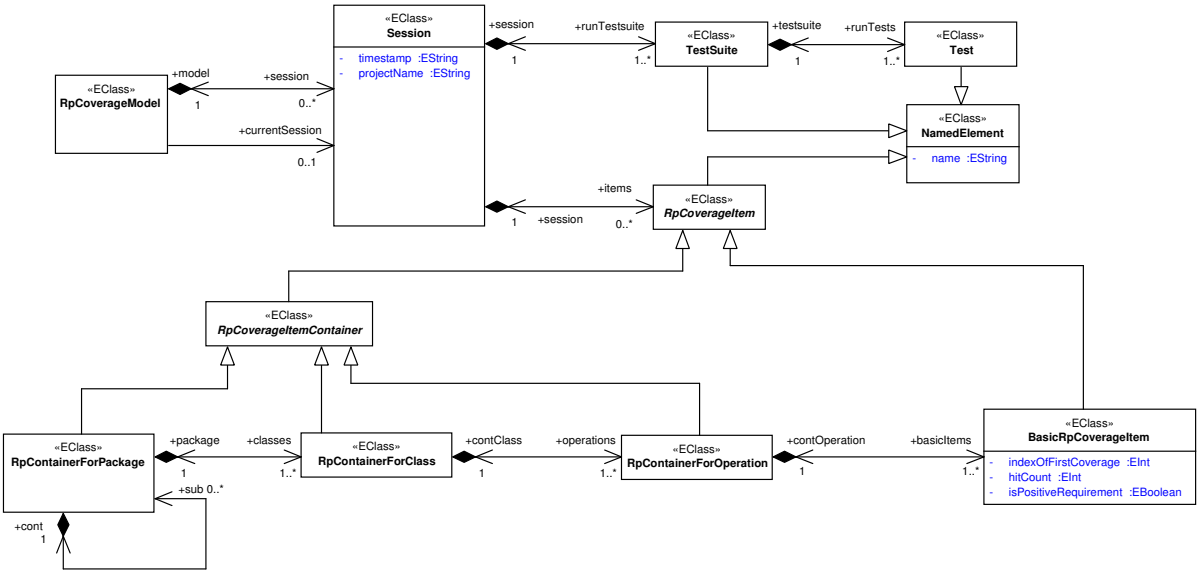


Abbildung 7.12: Metamodell zur Verwaltung, Serialisierung und Visualisierung der *RP*-Überdeckungswerte

`ForClass`) und Operationen (`RpContainerForOperation`) ab. Die Entsprechung der einzelnen abgeleiteten Coverage-Items ist die Klasse `BasicRpCoverageItem`. Letztere speichert in ihren Attributen den Index des ersten Tests, im Rahmen dessen die Testanforderung zum ersten Mal erfüllt wurde, und ob es sich um ein „positives“, i. S. v. nicht-negiertes, Coverage-Item handelt. Das Attribut `hitCount` ist zurzeit ungenutzt, bietet aber einen Hinweis auf eine nahe liegende Erweiterung des Testansatzes: statt nur zu messen, ob ein Coverage-Item überhaupt überdeckt wurde, könnte es auch interessant sein zu zählen, wie oft eine entsprechende Anforderung erfüllt wurde.

7.3.2 Test-Rümpfe

Wie bereits in der Beschreibung zur Abbildung 7.10 dargelegt, werden in Schritt 16 des Generierungsprozesses auch Test-Rümpfe bzw. -Stubs generiert. Hierbei handelt es sich um Java-Code in Form einiger Hilfsklassen, die es dem Benutzer ermöglichen, seine Tests leicht mit den für die Bestimmung der Überdeckung notwendigen Funktionen zu versehen und reproduzierbar und kontrolliert auszuführen sowie das Ergebnis zu visualisieren. Der generierte Code lässt sich funktional in zwei Teile unterteilen. Einerseits wird eine projektspezifische, erweiterbare Beispiel-Test-Suite für das JUnit-Rahmenwerk generiert, andererseits werden Hilfsklassen erstellt, welche die Kommunikation mit der *GUI*-Komponente bewerkstelligen und für kontrollierte Testbedingungen sorgen. Anhang C enthält in Abschnitt C.1 den Code der Beispiel-Test-Suite für das Transformationsbeispiel. Die Test-Suite bedarf nur sehr geringer manuellen Anpassungen in den Zeilen 10 und 11, Listing C.2, zur Einbindung der manuell erstellten Tests bedarf. Der Abschnitt C.2 bietet einen Exzerpt des generierten Codes für den zweiten Punkt, der für die Überdeckungsmessung bei der Ausführung der JUnit-Test-Suite sorgt.

Test-Suite

Die initial generierte JUnit-Test-Suite dient dazu, die JUnit-Tests, die hinsichtlich der durch sie erreichten Überdeckung bewertet werden sollen, gesammelt auszuführen und die Übermittlung der Überdeckungswerte an eine weitere Komponente sicherzustellen. Dazu stellt die Test-Suite sicher, dass vor der Ausführung der eigentlichen Tests in einer extra Laufzeitumgebung, diese für die Bestimmung der Überdeckung konfiguriert und initialisiert ist. So müssen sich Anwender nicht um diese technischen Details kümmern. Wie dem Listing C.1 in Zeile 9 zu entnehmen ist, umfasst dazu eine Haupt-Test-Suite eine einzelne innere Test-Suite – eine Instanz der Klasse `CustomizableTestSuite` – welche letztendlich durch den Benutzer anzupassen ist. Hierzu werden in den Zeilen 6 bis 11, Listing C.2, die Klasse(n) angegeben, welche die bereitzustellenden Testmethoden enthalten. Für das konkrete Beispiel sind dies die Klassen `Bd2JaConverterBasicTests` und `BdPreprocessorTests`, welche beide diverse Tests enthalten.

Von der RPC-Erfassung zur Visualisierung der Werte

Im Folgenden werden Details der Ausführung der Tests sowie der Anbindung der Eclipse-*GUI* skizziert. Dazu ist in Abbildung 7.13 die Situation bei der Testausführung dargestellt. Die beiden beteiligten Instanzen der *Java Virtual Machine (JVM)* sind mit JVM_1 und JVM_2 bezeichnet und sind als Regionen in der Abbildung eingezeichnet.

In JVM_1 läuft die Eclipse-*IDE*, welche den *Workspace* umfasst, der wiederum die zu testende Transformation sowie weitere benötigte Entwicklungsprojekte bereitstellt. Die zu testenden Transformation – hier die bereits vorgestellte Beispieltransformation – ist im BD2Ja-Projekt definiert, welches von dem Inhalt dreier weiterer Projekte unmittelbar abhängt, nämlich (i) BlockDiagramLanguage, dem Metamodellprojekt der Quellsprache, (ii) JavaLanguage, dem Metamodellprojekt der Zielsprache, sowie (iii) BD2JaRequirements, das Behelfsprojekt, das den aus *RPs* generierten Code umfasst. Bleibt noch das Projekt BD2Ja_Testsuite, welches stellvertretend für die mittels Überdeckung zu bewertenden Tests steht. Des weiteren umfasst Eclipse die wesentlichen Plugins, z. B. für den *RPC*-Ansatz oder JUnit, sowie die Anzeige der *GUI*. Die gestrichelten Pfeile sollen andeuten, dass die eingezeichnete Visualisierung der Überdeckung in diesem Fall durch das `SdmTestCoverageFramework`-Plugin und eines der `JUnit`-Plugins bereitgestellt wird, vgl. hierzu auch noch mal Abbildung 7.11.

Bei der Testausführung aus Eclipse heraus werden die JUnit-Tests in der zweiten *JVM* ausgeführt; dies is angedeutet durch den mit „Testsuite@Runtime“ überschriebenen Bereich. Dabei werden die Ergebnisse der Testausführung standardmäßig in Eclipse im entsprechende JUnit-*View* dargestellt. Dafür ist die zweite *JVM* durch eine JUnit-eigene *Socket*-Verbindung mit der Eclipse-Instanz gekoppelt. In Abbildung 7.13 ist diese durch das mit $\mathsf{Socket}_\mathrm{b}$ bezeichnete Element angedeutet. Andererseits wird eine zweite, hier als $\mathsf{Socket}_\mathrm{a}$ bezeichnete Verbindung genutzt, um die benötigten Überdeckungsinformationen zu übermitteln. Diese Verbindung wird vom `SdmTestCoverageFramework` auf Eclipse-Seite und den generierten Test-Rümpfen auf JUnit-Seite etabliert.

Die Basis hierfür wird in Listing C.1, Zeile 8, mit Hilfe der JUnit-eigene `@RunWith`-Java-Annotation gelegt, die festlegt, mit welcher `Runner`[15]-Instanz die Unit-Tests ausgeführt werden sollen. In diesem Fall kommt eine Instanz der in Listing C.3 definierten Klasse

[15] `org.junit.runner.Runner`

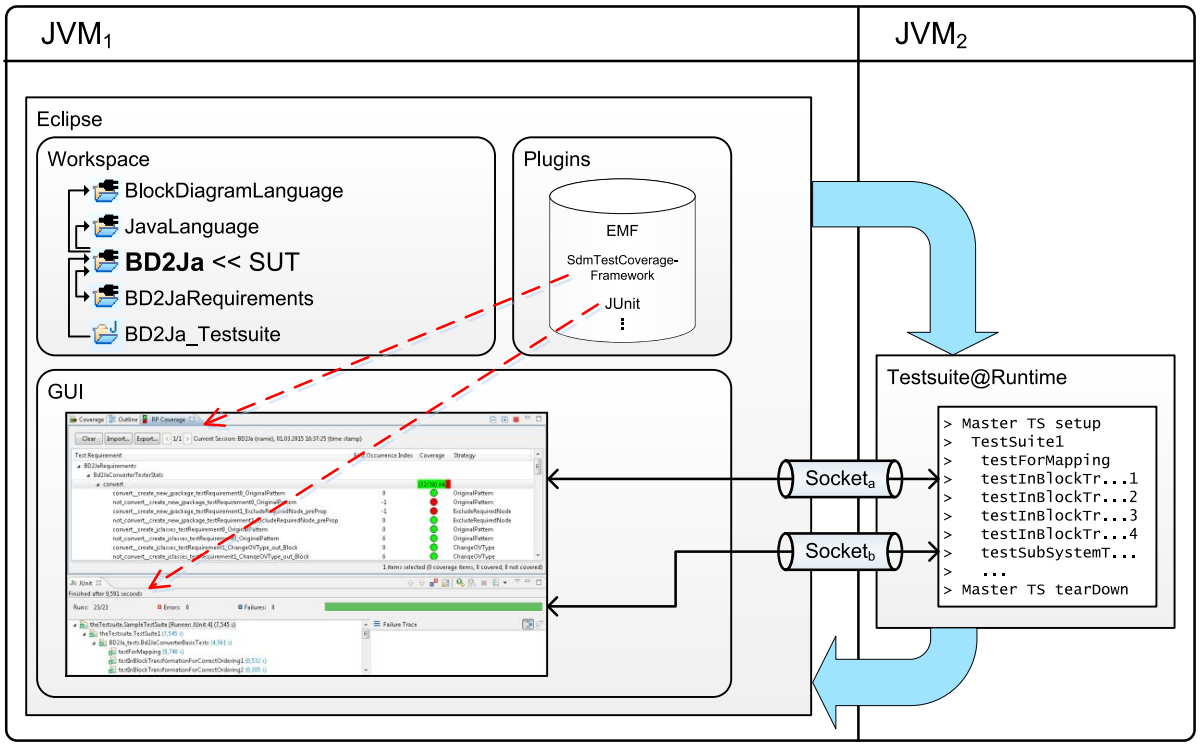


Abbildung 7.13: Anbindung der Anzeige (View) an die Testausführung

`CoverageReportingTestRunner` zum Einsatz, welche die JUnit-eigene Klasse `BlockJUnit4ClassRunner` erweitert. Die entsprechende Runner-Instanz erfüllt zwei zentrale Anforderung. Einerseits sorgt sie dafür, dass die Tests in einer deterministischen Reihenfolge ausgeführt werden, s. Methode `computeTestMethods()` ab Zeile 30). Andererseits stellt sie mit Hilfe der `TestRule`-Unterklasse `ReportingTestRule` sicher, dass vor der eigentlichen Testfallausführung, s. Zeil 60, ein entsprechendes Signal an die Auswertungs- und Visualisierungskomponente geschickt wird, s. Zeil 58. Daraufhin werden im Anschluss die Überdeckungsdaten übermittelt, vgl. Zeile 64. Abschließend erfolgt noch eine Signalisierung für den gesamten Test in Zeile 65. Zur Übermittlung der Nachrichten an das Plugin wird der in den Listings C.4 und C.5 angedeutete Kanal genutzt, welcher technisch auf Instanzen von `java.net.Socket` und `java.net.ServerSocket` aufbaut.

7.4 Anwendung am Beispiel

Nachdem der *RPC*-Ansatz aus unterschiedlichen Blickrichtungen beschrieben wurde, gilt es nun, sich den Fragen hinsichtlich seiner Anwendung zu widmen. Dabei liegt eine Anwendung im Umfeld das konkreten Beispiels aus Abschnitt 4.4 nahe. Zuvor soll aber auch noch kurz auf den grundsätzlichen Testprozess auf Basis der *RP*-Überdeckung aus Anwendersicht eingegangen werden.

7.4.1 Qualitätssicherung im Entwicklungssprozess

In Abbildung 7.14 ist ein einfacher Entwicklungs-, Test- und Korrekturprozess dargestellt. Die Abbildung stammt ursprünglich aus [WS13] und ist angelehnt an die Abbildung 5.2 aus [AO08, S. 181], einer Darstellung eines generischen Mutationstestprozesses. Für die Darstellung wird implizit angenommen, dass die Entwicklung des *SUT* der Entwicklung von entsprechenden Tests vorausgeht, also *kein* Test-First-Ansatz verfolgt wird. Dies erscheint schon alleine deswegen sinnvoll, da die *RP*-Analyse die Leistungsfähigkeit der vorhandenen Tests für das konkrete *SUT* zur *Ausführungszeit* bewerten soll.

Der Prozess startet mit der Implementierung einer initialen Version der zu testenden Transformation. Danach folgen mit den Schritten ① bis ④ die wesentlichen Schritte der *RP*-Generierung, wie bereits zuvor detailliert beschrieben. Nach Codegenerierung und Testfallerstellung können die Tests ausgeführt werden, vgl. hierzu die mit „Run tests“ bezeichnete Aktion. Sollten Tests zu Fehlern führen, so sind diese zu beheben. Dadurch ergibt sich eine neue, korrigierte Version der Implementierung. Hier schließt sich der obere Zyklus, in dem es um die Verbessung der Transformation geht, da anschließend für die neue Version wiederum Coverage-Items abgeleitet und Code generiert werden kann/muss. Führen die Tests allerdings zu keinen erkennbaren Fehlern, kann im Anschluss die erreichte Überdeckung in ⑤ sinnvoll ermittelt und mit einer in ⑥ gewählten Vorgabe in Schritt ⑦ verglichen werden. So kann entschieden werden, ob der tatsächliche Überdeckungsgrad als ausreichend erachtet wird. Ist dies nicht der Fall, sind fehlende Testfälle noch zu identifizieren und zu ergänzen, wodurch sich ein zweiter Zyklus ergibt, der eine Verbesserung der Testmenge zum Ziel hat.

Die Aktion ⑧ soll auf die Möglichkeit hindeuten, dass durch das Testen auch neue Invarianten aufgedeckt werden können, die entweder durch Tests zu überprüfen sind oder durch anderweitige Analysen nachgewiesen werden können. Falls Teile der generierten Anforderungen als nicht erfüllbar erkannt werden, sollten einerseits die Anforderungen von der Berechnung der Überdeckungsmetrik ausgeschlossen werden, um die tatsächliche Abdeckungsleistung nicht zu unterschätzen. Andererseits können aus solchen unerfüllbaren Anforderungen Zusicherungen abgeleitet werden, die beispielsweise im Rahmen von Regressionstests dazu benutzt werden können, um potentiell Hinweise auf neu eingebaute Fehler zu erhalten, falls die Zusicherungen unerwarteterweise ihre Gültigkeit verlieren.

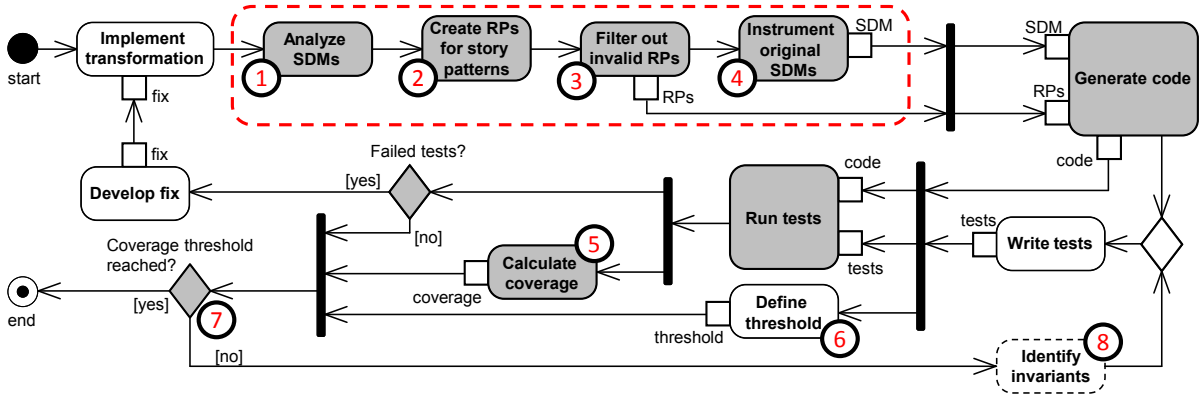


Abbildung 7.14: *RP*-basierter Test- und Korrekturprozess als UML-Aktivitätsdiagramm aus [WS13] (mit Grau hinterlegt bedeutet „automatisierbar“)

7.4.2 RPC-Auswertung für das Beispiel

Nachfolgend werden einige der Ergebnisse vorgestellt, die anhand der Anwendung der *RP*-Überdeckung auf die Beispieltransformation gewonnen werden konnten. Es werden drei Hauptaspekte betrachtet: (i) die grundsätzliche Anwendbarkeit des Verfahrens anhand der Anwendung auf die Transformation sowie initiale Daten zur Anzahl und Verteilung der erzeugten Coverage-Items, (ii) das Ziehen eines ersten Vergleichs zu klassischen, auf Quellcode basierenden Überdeckungskriterien sowie (iii) eine Untersuchung der zusätzlichen Laufzeitkosten des Verfahrens aufgrund des zu erwartenden Malus durch die zusätzliche *RPC*-Auswertung. Jedem Punkt ist im Folgenden ein eigener Unterabschnitt gewidmet.

Grundlegende Kenngrößen

Bei der Anwendung der *RP*-Generierung auf die Beispieltransformation, vgl. Anhang A.3, ergeben sich in Summe 319 abgeleitete *RPs* und entsprechend mit 638 aufgrund der Verdopplung durch die Forderung nach Erfüllung und Nicht-Erfüllung doppelt so viele Coverage-Items. Bei insgesamt 18 Operationen, die mit der *SDM*-Sprache implementiert wurden, entfallen so im Schnitt 35,4 Coverage-Items bzw. 17,7 *RPs* auf jede Operation. Allerdings unterscheiden sich die Umfänge der generierten Mengen an *RPs* von Operation zu Operation teils erheblich. Dies wird anhand der Abbildung 7.15a deutlich, die zwei Boxplots[16] für die Anzahl der generierten *RPs* pro Operation zeigt, jeweil einen für eine der beiden Teilfunktionalitäten der Gesamttransformation, welche jeweils innerhalb einer Klasse umgesetzt sind. Im linken Teil (Klasse `Bd2JaConverter`) ist die Streuung der Werte erkennbar größer als im rechten Teil (Klasse `BdPreprocessor`). Abgesehen von der etwas eingeschränkten Aussagekraft aufgrund der relativ kleinen Stichprobengröße liegen drei Hypothesen zu möglichen Ursachen der Beobachtung nahe:

1. Die Anzahl der Operationen ist im linken Fall größer, so dass auch Vertreter enthalten sind, die Ausreißer darstellen, allerdings noch nicht groß genug, um den Effekt solcher Ausreißern zu beschränken.

2. Der Bezug auf zwei Metamodelle (Ein- und Ausgabesprache) im Falle der Übersetzung könnte tendenziell vermehrt zu solchen Situationen führen, in denen insgesamt mehr Möglichkeiten bestehen, um *RPs* aus der *LHS* einer Regel abzuleiten. Unter anderem, weil auch die *LHSs* tendenziell mehr Elemente umfassen.

3. Für den konkreten Fall erfolgt die Implementierung der Übersetzung in einer Art Top-Down-Beschreibung (in Richtung vom Allgemeinen hin zum Speziellen), in der relativ viele Muster mit den allgemeinsten Typen aus den Metamodellen formuliert sind, was sich unmittelbar in vielen legalen Anwendungsstellen der *COT*- sowie der *CLTS*- und *CLTC*-Strategien niederschlägt.

Eine weitere interessante Kenngröße ist die Anzahl der abgeleiteten *RPs* für eine einzelne *LHS* einer *GT*-Regel. Betrachtet man die absolute Häufigkeit für das Auftreten

[16] Erstellt mit Hilfe der `boxplot(X)`-Funktion von Matlab unter Rückgriff auf die dort genutzten Standardeinstellungen; die Mittelmarkierung zeigt den Median, obere und untere Box-Grenzen entsprechend den Grenzen des 75%- und 25%-Quartils und die beiden Antennen/Whisker beschreiben die in den Messungen aufgetretenen Maximal- und Minimalwert in der Population.

der einzelnen Werte dieser Kenngröße für die insgesamt 84 Regeln[17] der 18 Operationen, so ergibt sich das Histogramm aus Abbildung 7.15c. Leicht zu erkennen ist, dass für die überwiegende Mehrheit der Regeln nur ein oder zwei *RPs* abgeleitet werden konnten. Bei näherer Betrachtung wird deutlich, dass die Hauptursache hierfür in den vielen kleinen Regeln liegt, die (a) eine isolierte Typüberprüfung darstellen (z. B. check_if_aSystem_is_Subsystem in Abbildung A.12), (b) ausschließlich den this-Knoten einführen oder (c) aber in dem Sinne minimal sind, dass sie jeweils nur eine ungebundene *LV* und *OV* enthalten, wobei letztere im Beispiel überwiegend von konkretem Typ und oft ohne weitere Unterklassen ist. Die Beobachtung, dass hier sehr häufig relative wenige *RPs* pro Regel abgeleitet werden, lässt sich folglich darauf zurückführen, dass bei dem hier genutzten Stil der *SDM*-Programmierung große und komplexere Muster viel seltener sind als kleine und einfache Muster. Letztere lassen aber auch tendenziell weniger Raum für Fehler, was die Mustersuche anbelangt. So kann man es als durchaus sinnvoll ansehen, dass das Überdeckungskriterium für das Testen solcher Regeln verhältnismäßig weniger Anforderungen produziert.

Darüber hinaus erscheint die dedizierte Untersuchung des hierdurch zu vermutenden Zusammenhangs zwischen der Größe der *LHS* einer Regel, im Sinne von Variablenanzahl in der linken Regelseite, und der Anzahl an abgeleiteten *RPs* aus der Regel interessant. In Abbildung 7.15b wird hierzu in einem Streudiagramm/Scatterplot die Anzahl der abgeleiteten *RPs* über der Größe der *LHS*, bezogen auf die Anzahl der Elemente, aufgetragen. Die resultierende Punktewolke lässt durchaus eine Interpretation dergestalt zu, dass wahrscheinlich ein Zusammenhang zwischen den beiden Größen besteht, welcher der ersten Intuition entspricht. Vereinfacht ausgedrückt, je umfangreicher die linke Seite der Regel, desto wahrscheinlicher resultieren daraus viele *RPs* für die Regel.

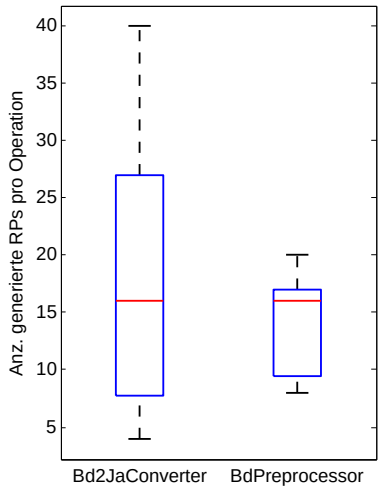
Vergleich zu Codeüberdeckungsmetriken

Die Entwicklung der Beispieltransformation aus Abschnitt 4.4 erfolgte testgestützt, so dass jede Teilfunktionalität als initial getestet anzusehen ist. Folglich existiert eine erste Testmenge aus JUnit-Tests, die manuell ad hoc entwickelt wurde und alleine auf der Erfahrung und der Intuition des Entwicklers (und somit mir selbst) basiert. Dabei wurde die Menge an Tests klassisch iterativ und parallel zur Erweiterung der Transformation entwickelt, wobei neue Funktionalitäten durch entsprechende neue Eingabemodelle stimuliert wurden. Als Orakel kommen manuell erstellte Zusicherungen zum Einsatz, die technisch als JUnit-Assertions realisiert sind. Werkzeuge zur Analyse einer irgendwie gearteten Form der Überdeckung kamen bei der Entwicklung der initialen Tests nicht zum Einsatz. Die resultierende Test-Suite umfasst einerseits 6 Testfälle für die Optimierungs- bzw. Normalisierungsschritte durch den Präprozessor `BdPreprocessor`, vgl. Abb. A.3 und Abschnitt A.3.2, andererseits 16 Testfälle für die Übersetzung von Blockdiagramminstanzen in Instanzen des Java-Metamodells durch den Translator `Bd2JaConverter`, vgl. Abschnitt A.3.3.

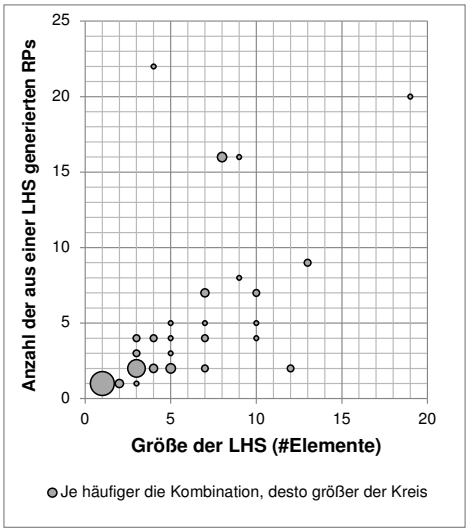
Auf Basis der zur Verfügung stehenden Testmenge wurden zwei Fragestellungen untersucht, nämlich:

1. Wie gut ist die vorhandene Testmenge, bezüglich des *RPC*-Ansatzes aber auch – da

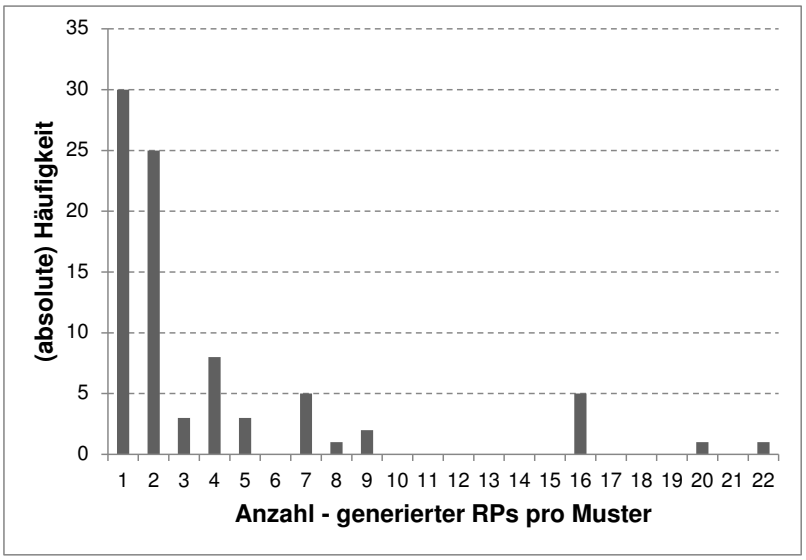
[17] Es wurden hierbei nur „echte Regeln“ mit nicht-leerer linken Seite berücksichtigt, für die mindestens ein *RP* generiert werden konnte.



(a) Boxplot mit Median, dem 25 %- resp. 75 %-Quartil und den beiden Extremwerten zur Anzahl der generierten *RPs*



(b) Streudiagramm zum Umfang einzelner *RPs*



(c) Histogramm zur Verteilung der Anzahl an generierten *RPs* pro Regel bzw. Muster

Abbildung 7.15: Statistiken zu den generierten *RPs* der Bd2Ja-Transformation

hier aufgrund des generativen Ansatzes Quellcode vorliegt – bezogen auf klassische Code-Überdeckungsansätze?

2. Ist ein Zusammenhang zwischen *RP*-Überdeckung und klassischen Überdeckungsmetriken erkennbar?[18] Und falls ja, wie deutlich tritt dieser zu Tage? Zur Erinnerung: Geiger z. B. argumentiert mit dem Vorhandensein eines solchen Zusammenhangs für die Nutzung codebasierter Überdeckungskriterien, vgl. [Gei11, S. 75].

Zur Beantwortung der ersten Frage wurden verschiedene Messungen zur Bestimmung der benötigten Überdeckungswerte durchgeführt. Zu Beginn wurden die *RPC*-Werte für das Beispiel bestimmt. Von den insgesamt 638 Coverage-Items wurden durch die vorhandenen Tests 413 abgedeckt, was einem Wert für die Metrik von 0,647 ergibt (entsprechend 64,7% Überdeckung). Betrachtet man die positiven bzw. negativen Anforderungen getrennt, so ergaben die Messungen ein Verhältnis von 180 zu 319, also 56,4% Abdeckung, für den ersten Fall und 223 zu 319 bzw. 73,0% Abdeckung für den zweiten Fall. Die Werte erscheinen relativ niedrig und sind sehr wahrscheinlich als ausbaufähig zu betrachten, wenn auch hierzu noch genauere Untersuchungen notwendig sind. Auf Optimierungsschritte der Testmenge hinsichtlich höherer Überdeckungswerte soll an dieser Stelle aber verzichtet werden, da entsprechenden Fragestellungen im Rahmen der eigentlichen Evaluation, vgl. Kapitel 9, dediziert nachgegangen wird.

Neben den *RPC*-Werten wurden auch Messungen zur (i) Anweisungsüberdeckung (Instruction-Coverage), zur (ii) Zeilenüberdeckung (Line-Coverage) sowie zur (iii) Verzweigungsüberdeckung (Branch-Coverage) durchgeführt. Zur Bestimmung der Werte hinsichtlich Punkt (i) wurde das *EclEmma*-Plugin[19] für Eclipse, basierend auf der *Java Code Coverage Library (JaCoCo)*, in Version 2.3 benutzt. Die Messwerte für (ii) und (iii) wurden mit Hilfe von *Cobertura*[20], Version 2.0.3, bestimmt. In Tabelle 7.2 sind die entsprechenden Messwerte dargestellt, aufgeschlüsselt nach Operation und Überdeckungskriterium. Bezüglich der Messungen wurden zwei Codeversionen unterschieden, nämlich (a) einmal mit den Codeanteilen der *RPC*-Instrumentierung sowie (b) ohne diese. (Für den zweiten Fall ist eine *RPC*-Bestimmung unmöglich.) Je dunkler bzw. rötlicher einer Zelle in der Tabelle eingefärbt ist, desto niedriger ist der entsprechende Wert der Abdeckung und je heller bzw. grünlicher, desto höher ist der Wert.

Analysiert man die Ergebnisse genauer, so fällt sofort auf, dass die Werte für Anweisungs- und Zeilenüberdeckung einerseits nicht exakt übereinstimmen, andererseits aber vergleichbare und hohe Werte erreicht werden, insbesondere für den Fall (b). Ersteres deutet auf unterschiedliche Arten der Bestimmung der Überdeckungswerte hin, zweiteres auf den eingeschränkten Nutzen dieser Überdeckungskriterien zum Testen (nicht nur) von *SDM*-basierten Implementierungen. Die Werte für *RPC* sowie für die Verzweigungsüberdeckung sind dagegen relativ niedrig, was jeweils auf fehlende Tests und nicht ausreichend gründliches Testen hindeutet. Interessant ist dabei vor allem die Beobachtung, dass sich genau an den Stellen mit den größten Deltas zwischen den Fällen (a) und (b), bezogen auf die Verzweigungsüberdeckung, auch hinsichtlich der *RP*-Überdeckung die mit Abstand größten Abdeckungswerte einstellen.

[18] Die Beantwortung der Frage kann hierbei selbstverständlich nicht allgemein erfolgen. Sie ist Gegenstand eigenständiger Forschung, vgl. beispielsweise [Bar+03; Kir09; Eri+12].

[19] `http://www.eclemma.org/` (abgerufen am 21.3.2014)

[20] `http://cobertura.github.io/cobertura/` (abgerufen am 21.3.2014)

Aufgrund dessen lässt sich die Hypothese aufstellen, dass die beiden Überdeckungsarten, *RPC* auf *SDM*-Ebene sowie Verzweigungsüberdeckung auf Code-Ebene, korreliert sein könnten. Um dies eingehender zu untersuchen, wurden entsprechende Streudiagramme angefertigt, die jeweils die *RPC*-Werte den Codeüberdeckungswerten gegenüberstellen. In der Abbildung 7.15 sind die resultierenden Diagramme abgebildet. Für jede der Punktewolken sind auch die Regressionsgraden und das zugehörige Bestimmtheitsmaß R^2 enthalten.[21]. Es lässt sich insbesondere aus Teilabbildung 7.16b ablesen, dass eine solche Korrelation wahrscheinlich ist. Im Fall (b) lässt sich auch ein Zusammenhang zwischen *RPC* und Zeilenüberdeckung erkennen, was angesichts der vermuteten Korrelation zwische *RPC* und Bedingungsüberdeckung einerseits sowie der Subsumptionbeziehung zwischen Bedingungs- und Zeilenüberdeckung andererseits nicht ausgeschlossen erscheint.

	RPC			Instruction-Coverage (EclEmma)		Line-Coverage (Cobertura)		Branch-Coverage (Cobertura)	
	(a)			(a)	(b)	(a)	(b)	(a)	(b)
	comb.	pos	neg						
Bd2JaConverter									
convert	84.2%	78.9%	89.5%	85.0%	92.9%	76.5%	93.8%	61.7%	64.6%
convertSystem	70.0%	63.3%	76.7%	82.3%	91.6%	75.2%	95.2%	61.3%	63.5%
createSystemToMethodMappings	59.4%	68.8%	50.0%	79.7%	91.4%	69.1%	93.8%	60.0%	63.6%
embedInContainerExpr	59.6%	96.2%	23.1%	70.2%	81.4%	64.3%	90.3%	59.5%	62.5%
establishJParamOrdering	86.7%	86.7%	86.7%	76.0%	87.7%	70.8%	94.7%	68.0%	70.9%
getTopmostAssignment	56.3%	50.0%	62.5%	65.6%	83.0%	59.6%	88.2%	54.2%	56.3%
init	85.7%	92.9%	78.6%	65.6%	89.5%	64.3%	93.0%	67.3%	75.0%
visitAddBlock	52.5%	27.5%	77.5%	75.5%	87.3%	65.5%	88.8%	55.4%	56.8%
visitBlock	78.6%	85.7%	71.4%	71.7%	85.8%	61.5%	89.9%	65.4%	83.3%
visitConstantBlock	50.0%	75.0%	25.0%	74.4%	87.7%	67.0%	90.0%	53.3%	54.2%
visitInBlock	50.0%	75.0%	25.0%	74.1%	87.8%	66.7%	90.0%	53.6%	54.5%
visitMultBlock	50.0%	25.0%	75.0%	75.6%	87.6%	65.7%	88.9%	55.6%	57.1%
visitSubSystemBlock	50.0%	26.9%	73.1%	80.0%	89.2%	71.8%	93.1%	55.4%	56.5%
BdPreprocessor									
collectRelevantBlocks	77.5%	60.0%	95.0%	80.1%	90.4%	73.3%	91.8%	71.1%	75.0%
normalizeGain	62.5%	37.5%	87.5%	85.9%	93.3%	83.8%	95.1%	61.1%	62.5%
process	90.0%	90.0%	90.0%	73.7%	90.7%	63.6%	94.4%	60.4%	66.7%
splitAdd	68.8%	50.0%	87.5%	87.1%	94.4%	84.8%	96.0%	77.9%	79.7%
splitMult	68.8%	50.0%	87.5%	87.1%	94.4%	84.8%	96.0%	77.9%	79.7%

Tabelle 7.2: Resultate verschiedener Überdeckungsmessungen

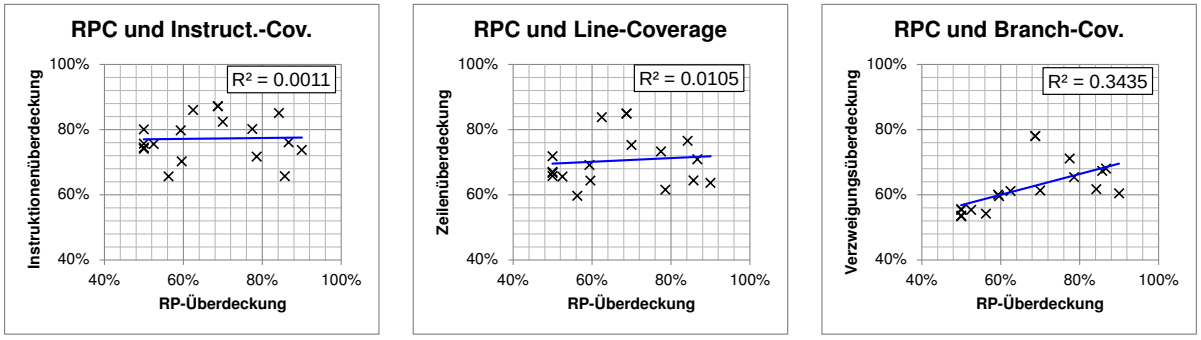
Laufzeitaspekte der RPC-Auswertung

Kommen wir nun zu dem dritten der Aspekte des *RPC*-Einsatzes am Beispiel, nämlich der Laufzeit sowie möglicher negativer Einflüsse auf diese durch die Instrumentierung und die zusätzlichen Pattern-Matching-Schritte bei der *RP*-Auswertung. Hierzu wurden die Laufzeiten der Ausführung für die vorhandene Test-Suite in mehreren Situationen gemessen und miteinander verglichen.

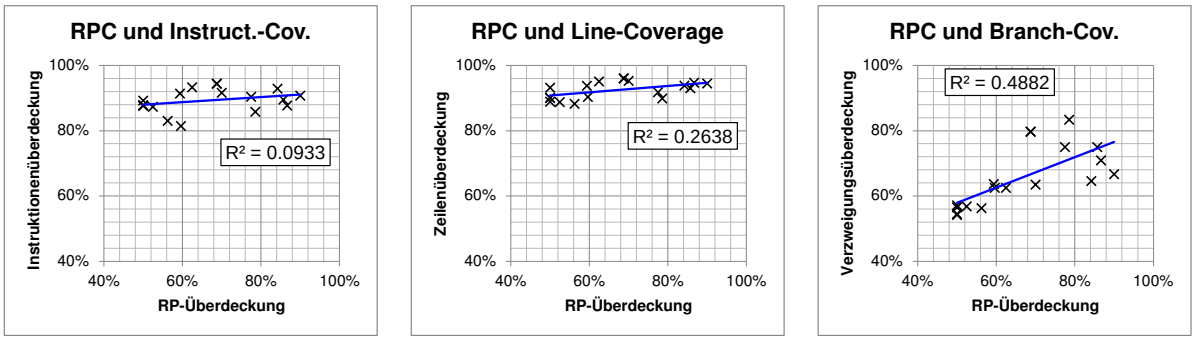
In Abbildung 7.17 sind die jeweils über 10 Einzelmessungen gemittelten Laufzeiten in Sekunden für die fünf zugrunde gelegten Szenarien dargestellt. Im Rahmen der Messungen wurden die folgenden Szenarien, der Reihenfolge entsprechend von links nach rechts in der Tabelle, untersucht:

1. Situation: der (Byte-)Code ohne Instrumentierung unter Verwendung des Standard-(JUnit-Test-)Runners.

[21] Der Korrelationskoeffizient ρ nach Bravais und Pearson, ergibt sich hier durch Wurzelbildung aus dem Bestimmtheitsmaß ($\rho = \sqrt{R^2}$).



(a) Streudiagramme für Situation „mit SDM-Instrumentierung“ (bezogen auf den Java-Code)



(b) Streudiagramme für Situation „ohne SDM-Instrumentierung“ (bezogen auf den Java-Code)

Abbildung 7.16: Untersuchung möglicher Korrelationen (auf Ebene einzelner Operationen) zwischen den Überdeckungsansätzen (R^2 entspricht dem Quadrat des Korrelationskoeffizienten)

2. Situation: der (Byte-)Code ohne Instrumentierung unter Verwendung einer Instanz der `CoverageReportingTestRunner`-Klasse (im Folgenden als *Reporting-Runner* abgekürzt).

3. Situation: der mittels *Cobertura* offline[22] instrumentierte Bytecode unter Verwendung des Standard-Runners.

4. Situation: der mittels *EclEmma* (genauer mittels *JaCoCo*) zur Laufzeit „on-the-fly" instrumentierte (Byte-)Code unter Verwendung des Standard-Runners.

5. Situation: die auf *SDM*-Ebene instrumentierte Implementierung, bei der sich der generierte Java-Code von dem zuvor verwendeten Java-Code entsprechend unterscheidet; dabei kommt der Reporting-Runner zum Einsatz.

Hinsichtlich der Messwerte fallen zwei Aspekte besonders auf. Zum einen ist alleine durch den Einsatz des Reporting-Runners, selbst im Falle ohne jede Instrumentierung, bereits eine deutliche Verlangsamung bezüglich der Ausführungszeit zu beobachten. Offenkundig beeinflussen die Laufzeiteigenschaften der Ein- und Ausgabe aufgrund der Socket-Verbindung die Gesamtlaufzeit erheblich. Der Effekt des zusätzlichen Pattern-Matchings aufgrund der generierten *RPs* tritt hierdurch in den Hintergrund (vgl. den zweiten Balken von links mit dem äußersten rechten Balken) und ist nur im Falle der Tests für die Teiltransformation des `BdPreprocessor` überhaupt zu erkennen. Zum anderen bewegt sich der Laufzeitmehraufwand aufgrund der *RPC*-Instrumentierung – für dieses kleine Beispiel mit seinen relativ überschaubaren Testmodellgrößen[23] – in einer vergleichbaren Größenordnung, wie beispielsweise der Mehraufwand durch die vorhandenen Instrumentierungsanweisungen im Falle der Cobertura-Instrumentierung. Dies ist insbesondere deswegen interessant, da sowohl bei Cobertura als auch bei *RPC* die Instrumentierung offline erfolgt und die Ansätze diesbezüglich als konzeptionell vergleichbar anzusehen sind.

Dennoch bleibt festzuhalten, dass die *RPC*-Auswertung mit einem größeren Auswertungsaufwand einhergeht als bei klassischen strukturellen Code-Überdeckungsansätzen. Der hier zu beobachtende Laufzeitmehraufwand ist allerdings durchaus noch als vertretbar anzusehen, zumal die Überdeckung nicht bei jeder Testausführung erneut ermittelt werden muss, sondern dediziert nur für solche Fälle, für die Messwerte benötigt werden und noch keine erfolgreiche Überdeckung erkannt wurde. Für große Transformationen bleibt aber noch gesondert zu untersuchen, ob und ggf. inwiefern dies ebenso gilt. Falls hohe Ausführungszeiten tatsächlich zu einem Problem werden, könnte eine nahe liegende Optimierung darin bestehen, das kontinuierliche Übermitteln von Überdeckungswerten an die *GUI*-Komponente zur Laufzeit zu unterlassen und nur noch einmalig die Messwerte am Ende zu übermitteln. Allerdings würde dies keinen Einfluss auf die Herausforderungen aufgrund der vermehrten Anzahl an Schritten zur Mustersuche haben.

[22] Die Instrumentierung erfolgt auf Basis der zuvor kompilierten `.class`-Dateien per Batch-Verarbeitung unabhängig von der eigentlichen Ausführung.

[23] Die Größe eines Modells kann einen gravierenden Einfluss auf den zu erwartenden Aufwand für das *GT*-Pattern-Matching haben.

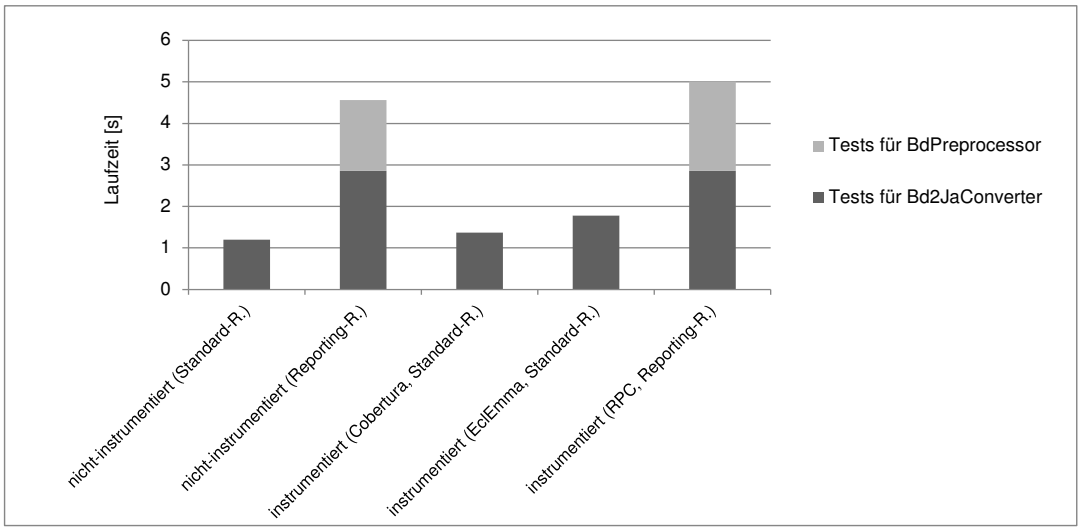


Abbildung 7.17: Einfluss der Instrumentierung auf die Laufzeiten der Testausführung

7.5 Bewertung und Zusammenfassung

In diesem Kapitel wurde das auf Graphmustern basierendes Überdeckungsmaß der *RP*-Überdeckung, kurz *RPC*, für programmierte Graphtransformationen am Beispiel der *SDM*-Sprache eingeführt und beschrieben (die zugrunde liegende Idee wurde bereits in [WS13] veröffentlicht). Dazu wurde eine Ableitungsvorschrift zur Erzeugung der benötigten Graphmuster in Form eines Algorithmus eingehend vorgestellt und auf verschiedene Strategien, die diesen ergänzen und konkretisieren, eingegangen. Darüber hinaus wurden technische Aspekte einer Implementierung und der Automatisierung wesentlicher Teilabläufe – insbesondere mit Hinblick auf mögliche Anwender – in Form eines Eclipse-Features, also einer Menge von Plugins, vorgestellt und zentrale Design-Entscheidungen dargelegt. Auf Basis der im Rahmen dieser Dissertation entstandenen Implementierung des *RPC*-Ansatzes wurde für die Beispieltransformation, vgl. Kapitel 4.4 bzw. Anhang A.3, über erste Ergebnisse hinsichtlich einer praktischen Anwendung berichtet.

7.5.1 Ein Zwischenfazit

An dieser Stelle sollen noch einmal kurz die Anforderungen aus dem Abschnitt 7.1 aufgegriffen werden. In der Tabelle 7.3 sind hierzu die zuvor aufgestellten acht Anforderungen referenziert und dabei für jede Anforderung vermerkt, ob sie durch den in diesem Kapitel vorgestellten Ansatz umgesetzt sind.

Da der Ansatz auf Graphmustern basiert und diverse Variationen der ursprünglichen Musterteile der Regeln erzeugt, erfüllt er Anforderung 7.1 nach der Berücksichtigung der Struktur von *GT*-Mustern unmittelbar. Es handelt sich ebenso offensichtlich auch um einen strukturellen Überdeckungsansatz, bei dem Implementierungsdetails ausgenutzt werden, wie es Anforderung 7.2 fordert. Durch die Art der Auswertung der *RPs* und die Position der Instrumentierungsanweisungen im Kontrollfluss sowie aufgrund der Tatsache, dass die *RPs* seiteneffektfrei ausgewertet werden können, wird auch Anforderung 7.3 erfüllt. Durch Anforderung 7.4 wird gefordert, dass der Ansatz Testanforderungen mit Bezugnahme auf die Operationen des Metamodells gruppiert und formuliert. Die abge-

	Anforderung							
	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8
Durch *RPC* erfüllt?	✓	✓	✓	✓	✓	(✓)	?	(✓)

Tabelle 7.3: Status der Umsetzung der Anforderungen aus Abschnitt 7.1

leiteten Coverage-Items, in Form der generierten *RPs*, beziehen sich zwar auf die Regeln als kleinste Einheit des Testens, werden aber durch die Art der visuellen Repräsentation, vgl. Abbildung 7.11, und durch das Metamodell aus Abbildung 7.12 für den Benutzer unterhalb der Operationen gruppiert. Somit ist diese Anforderung ebenfalls erfüllt. Anforderung 7.5 besagt, dass ein *SDM*-Testkriterium nicht ausschließlich über den Kontrollfluss definiert sein sollte, was von *RPC* offensichtlich auch erfüllt wird.

Die in Anforderung 7.6 geforderte Subsumption von Knoten- und/oder Kantenüberdeckung, bezogen auf den Kontrollfluss der *SDM*, ist in der hier beschriebenen Ausbaustufe des *RPC*-Ansatzes noch nicht vollständig gegeben und deshalb der Haken in der Tabelle entsprechend eingeklammert. Auf den ersten Blick scheint aufgrund der *UC*-Strategie und der resultierenden einfachen Kopie der *LHS* jeder Regel im Zusammenspiel mit der Forderung nach positiver und negativer Auswertung dieser Kopie, sichergestellt zu sein, dass jede Kante des Kontrollflusses traversiert wird. Allerdings wird hierbei nicht beachtet, dass in der hier verwendeten Version der Story-Diagramme auch anhand des Rückgabewertes von *Statement-Knoten* bedingt verzweigt werden kann. Da Statement-Knoten aber bisher in Bezug auf die *RPC*-Generierung ignoriert werden, ist durch Gesamtheit der *RPC*-Forderungen erst einmal nicht sichergestellt, dass auch alle ausgehenden Kanten eines solchen Knotens beim Testen traversiert werden. Besitzt der Knoten nur eine ausgehende Kante, wird diese bei einem Durchlauf selbstverständlich immer traversiert. Ansonsten gilt, dass, sollten sich auf allen möglichen Pfaden (entlang der gerichteten Kontrollflusskanten) von einem solchen Knoten zu jedem erreichbaren Endknoten mindestens ein *Story-Knoten* befinden, dies aufgrund der abgeleiteten Coverage-Items sicherstellt, dass alle jeweiligen Pfade bis hin zur Auswertung der *RPs* beschritten werden sollten. In Konsequenz führt dies wiederum dazu, dass alle ausgehenden Kanten des *Statement-Knotens* überdeckt werden. Da dies aber im Allgemeinen nicht immer gegeben ist, sollte der *RPC*-Ansatz diesbezüglich robuster gemacht werden; in allen möglichen Fällen sollten die ausgehenden Kanten der Statement-Knoten überdeckt werden, so dass unabhängig von den zuvor beschriebenen Einschränkungen die geforderte Subsumption garantiert ist. Hierzu wäre es notwendig, entsprechende Instrumentierungsanweisungen in die beiden ausgehenden Kontrollflusskanten einzubauen, was die Beobachtbarkeit der beiden möglichen Ausgänge ermöglichen würde und so eine entsprechende Forderung zur Abdeckung dieser gewährleistet.

Zur Beantwortung der Frage, ob Anforderung 7.7 durch den Ansatz erfüllt wird, bleibt festzuhalten, dass es durch die verschiedenartigen Forderungen aufgrund der *RPs* wahrscheinlich ist, dass verschiedenartige Testmodelle gefordert werden. Ob diese allerdings *ausreichend* verschiedenartig sind, in dem Sinne, dass damit typische Fehler mit hoher Wahrscheinlichkeit entdeckt werden, lässt sich bisher nicht abschließend beantworten. Dazu ist eine entsprechende Untersuchung anhand größerer Transformationen notwendig. Kapitel 9 reicht eine entsprechende Betrachtung nach.

Anforderung 7.8 ist bezogen auf die *RP*-Generierung erfüllt, da diese keine Anforderung an die Operationalisierung bzw. Ausführung oder Interpretation der Muster und Regeln stellt. Allerdings sind die Instrumentierungsanweisungen zurzeit technisch so formuliert, dass sie sich die Java-Codegenerierung zu Nutze macht. Dies ist keine zwingende Notwendigkeit, vereinfachte aber die Umsetzung im Rahmen der Implementierung. Eine spezielle Unterstützung für entsprechende Instrumentierungsanweisungen, beispielsweise innerhalb eines *SDM*-Interpreters, stellt konzeptionell kein Problem dar.

7.5.2 Ausblick und schließende Bemerkungen

Für den vorgestellten *RPC*-Ansatz konnte bis hier hin gezeigt werden, dass er praktisch umsetzbar ist. Daneben deuten die vorgestellten Ergebnisse der Experimente darauf hin, dass der Ansatz dazu geeignet ist, dabei zu helfen, die Qualität von *SDM*-Transformationen zu steigern. Für eine aussagekräftige Bewertung muss der Ansatz darüber hinaus allerdings noch systematisch evaluiert werden. Für eine systematische Evaluation des Testansatzes bietet sich eine *Mutationsanalyse* auf Basis der *SDM*-Beschreibung an. Das nachfolgende Kapitel ist deshalb der Beschreibung eines solchen gewidmet. Im Laufe der Beschreibung von *SDM*-spezifischen Mutationen werden gewisse Parallelen zur Ableitung der *RPs* erkennbar, da bei beiden Abläufen, ausgehend von der ursprünglichen Transformationen, durch systematische Änderungen neue Artefakte entstehen. Die Art und Weise, wie diese Variationen erzeugt werden, ähneln sich. Allerdings sind Zweck und Interpretation der Variationen sehr unterschiedlich. Dieser Aspekt wird im nächsten Kapitel noch deutlicher herausgearbeitet.

Bezogen auf mögliche Erweiterungen des *RPC*-Ansatzes erscheinen verschiedene Optionen reizvoll. Zum einen bietet die Entwicklung und Untersuchung weiterer Strategien zur Ableitung von *RPs* ein weites Feld. Auch die mögliche Verknüpfung mehrerer Muster zu komplexeren Prädikaten erscheint vielversprechend, insbesondere im Hinblick auf etablierte Verfahren des kombinatorischen Testens, wie beispielsweise *Pair-Wise*-Verfahren [WP01]. Für eine bessere Kontrollierbarkeit während der Auswertung solcher zusammengesetzter Muster wäre allerdings ein explizites Verwalten und Vorgeben von Teilmatches sinnvoll, da ansonsten der inhärent vorhandene Nichtdeterminismus bei der Mustersuche absolute Aussagen erschwert bis verhindert. Ein ähnliches Problem erwähnen auch Baldan et al. beim Testen auf Basis nichtdeterministischer Spezifikationen auf Grundlage von *GT*-Regeln in [BKS04].

Ein weiteres potentiell relevantes Feld liegt in der Untersuchung von Anknüpfungspunkten zwischen dem *RPC*-Verfahren und Methoden der formalen Verifikation. So könnten ggf. Model-Checking-Ansätze oder Methoden zur symbolischen Ausführung genutzt werden, um Testmodelle automatisiert zu generieren, die einen gewissen Grad der Abdeckung sicherstellen. Andererseits könnten in umgekehrter Richtung durch das Testen und die Entwicklung geeigneter Testfälle Erkenntnisse gewonnen werden, die für eine formale Verifikation bestimmter Eigenschaften der Transformation nutzbar sind. Anforderungen, die sich auch nach intensiver Beschäftigung als praktisch nicht erfüllbar herauskristallisieren sind beispielsweise potentielle Kandidaten, um in negierter Form als Invarianten genutzt zu werden – ggf. kann bei Bedarf anderweitig sogar die Unerfüllbarkeit bewiesen werden.

> [. . .] good tests are those that fail; a successful test is of dubious value [. . .]
>
> *(Gannon et al., aus [GMH81])*

8 Mutationsanalyse bei SDM-Transformationen

Nachdem im vorangegangen Kapitel das *RP*-Konzept für *SDM*-Transformationen vorgestellt wurde sowie die Anwendbarkeit anhand der Beispieltransformation gezeigt werden konnte, stellt sich jetzt die Frage nach der Bewertung der Leistungsfähigkeit des Ansatzes bezogen auf die Fähigkeit, tatsächlich zur Erkennung vorhandener Fehler beizutragen. Folglich soll die Frage beantwortet werden, ob Testmengen, die anhand des *RPC*-Kriteriums aus Kapitel 7 entwickelt wurden, in der Lage sind, Fehler mit hoher Wahrscheinlichkeit aufzudecken. Diese Eigenschaft sollte objektiv und quantifizierbar bewertet werden können, so dass beispielsweise angegeben werden kann, wie viele bekannte Fehler tatsächlich auch entdeckt werden konnten. Letzteres ist für tatsächliche Fehler nicht zu beantworten, da sich deren Existenz der Kenntnis entzieht. Ziel des Testens liegt ja darin, unbekannte Fehler zu finden und zu korrigieren.

Eine Möglichkeit, diesen Nachweise auf der Basis empirischer Untersuchungen durchzuführen, liegt darin, den Überdeckungsansatz im Rahmen von Feldstudien bei möglichst vielen Entwicklungsprojekten über einen längeren Zeitraum zu nutzen, um im Anschluss ein vergleichendes Fazit unter Berücksichtigung alternativer Verfahren zu ziehen. Dieser Ansatz sprengt allerdings den Rahmen des Möglichen innerhalb dieser Arbeit, da (i) zu wenige Ressourcen (zeitlich, personell) zur Auswertung vorhanden waren, (ii) der Nutzerkreis der *SDM*-Sprache zu eingeschränkt ist, und (iii) Erkenntnisse zu alternativen Methoden dergestalt ebenfalls fehlen. Darüber hinaus sind andere Ansätze zur Untersuchung grundsätzlich vorzuziehen, falls bei solchen ein höherer Grad an Reproduzierbarkeit und Kontrollierbarkeit bzgl. der Erkenntnisse zu erwarten ist.

Das in Kapitel 5.4 vorgestellte *mutationsbasierte Testen* ist ein besser geeignetes Hilfsmittel, um die Leistungsfähigkeit von Testansätzen wissenschaftlich bewerten zu können, vgl. z. B. [And+06]. Hierbei werden auf kontrollierte Art und Weise einzelne typische Fehler in das zu testende Programm implantiert und anschließend untersucht, ob die

vorhandene Testmenge ausreicht, um diese zu entdecken. Ist dies der Fall, so ist die Testmenge als ausreichend gut anzusehen, andernfalls sind die vorhandenen Tests noch nicht ausreichend. Im Folgenden wird für ein entsprechendes Vorgehen ein erweiterbares *Mutationsrahmenwerk* für die *SDM*-Sprache vorgestellt, mit dessen Hilfe die Erkennungsleistung für *SDM*-typische Fehler im Rahmen einer Evaluation untersucht werden kann.

Das Kapitel gliedert sich in sechs Unterabschnitte. Zuerst werden in Abschnitt 8.1 die größten Herausforderungen einer Entwicklung eines solchen Mutationsrahmenwerks und einer darauf aufbauenden initialen Komponente aufgeführt. Daraus werden dann in Abschnitt 8.2 konkrete Anforderungen an ein Mutationsrahmenwerk formuliert. Anschließend werden in Abschnitt 8.3 die umgesetzten Mutationen und in Abschnitt 8.4 Aspekte der Implementierung im Rahmen einer Eclipse-Erweiterung erläutert. Nachdem in Abschnitt 8.5 die Anwendung anhand des konkreten Beispiels skizziert wurde, folgt mit Abschnitt 8.6 eine kurze Bewertung des umgesetzten Mutationsansatzes.

8.1 Herausforderungen

Um die Leistungsfähigkeit des Überdeckungskonzeptes möglichst objektiv beurteilen zu können, werden größere Mengen fehlerhafter Spezifikationen benötigt. Gegen eine manuelle Erzeugung solch fehlerhafter Spezifikationen sprechen mehrere Argumente:

- Sehr großer Aufwand und damit teuer,
- Potentieller Bias durch die ausführenden Personen (z. B. durch Erwartungshaltungen hinsichtlich der anzutreffenden Fehler),
- Gefahr der fehlenden Systematik,
- Reproduzierbarkeit ist schlecht bzw. nicht gegeben,
- Programmänderungen sind monotone und wenig kreative Tätigkeiten,
- Potentielle Fehlerquelle Mensch.

Auf eine Automatisierung kann deshalb praktisch nicht verzichtet werden, s. dazu auch [AO08, S. 273].

Existierende Mutationswerkzeuge, wie z. B. *Mothra* [DeM+88], *muJava* [MOK05], *Jester* [Moo01], *Jumble* [Irv+07], *Javalanche* [SZ09], *PIT*[1] oder *CREAM* [DS07], wurden für klassische Programmiersprachen entwickelt. Sie lassen sich nicht gut auf Graph-Transformationsprogramme anwenden [MBT06a], da (i) nicht immer Code generiert wird, (ii) die Fehler auf einer zu technischen Ebene stattfinden, so dass entweder Annahmen der Semantik verletzt werden oder die Fehler an irrelevanten Stellen eingebaut werden – das Abstraktionsniveau ist insgesamt also falsch – und (iii) die eingebauten Fehler solchen entsprechen sollten, die ein Entwickler typischerweise auch bei der Entwicklung machen könnte. Bei der *SDM*-Entwicklung werden typische Java-Fehler z. B. eher selten auftreten, da einerseits der Codegenerator für eine konstante Codequalität sorgt und andererseits die direkte Verwendung von Java-Fragmenten in *SDM*-Diagrammen schlechten Stil darstellt und durch neue Konstrukte zunehmend in den Hintergrund tritt.

[1] `http://pitest.org/` (abgerufen am 22.4.2014)

8.2 Grundlegende Anforderungen

Ausgehend von den bisherigen Aussagen lassen sich Anforderungen an das zu entwickelnde Mutationsrahmenwerk formulieren. Direkt ersichtlich ist beispielsweise die weiter unten formulierte Anforderung 8.1. Sie folgt einerseits unmittelbar aus ökonomischen Überlegungen heraus, da ein günstiges Verhältnis von Kosten zu Nutzen dafür spricht, häufige Fehlerarten bevorzugt zu berücksichtigen. Andererseits stellt der Bezug zum menschlichen Faktor sicher, dass zu synthetisch anmutende Fehler, die entweder unwahrscheinlich sind, oder – beispielsweise durch *statische Analysen* – sogar unmittelbar auffallen würden, nicht im Fokus der Betrachtung liegen. Insgesamt ist darauf zu achten, dass die Aussagekraft der auf dem Mutationsrahmenwerk basierenden Ergebnisse nicht durch die Wahl von zweifelhaften Fehlerarten beschädigt wird.

Anforderung 8.1

Das Rahmenwerk soll überwiegend typische *Fehler in ein SDM-Programm implantieren, insbesondere aber solche, die auch ein menschlicher Entwickler realistischerweise gemacht haben könnte.*

Eine zweite Anforderung ergibt sich mit Anforderung 8.2 dahingehend, dass nicht alle Fehler von gleichem oder ähnlichem Typ sein sollten. Motivation hierfür ist, dass eventuell mögliche Verzerrungen aufgrund einer einseitigen Beschränkung auf bestimmte Fälle, beispielsweise durch eine selektive Auswahl, von Beginn an ausgeschlossen werden sollen. Vordergründig scheinen sich die Anforderungen 8.1 und 8.2 ein Stück weit zu widersprechen. Tatsächlich sollen beide Anforderungen in Kombination dafür Sorge tragen, dass sich weder eine Situation ergibt, in der ausschließlich ein einziger häufig auftretender Fehlertyp unterstützt wird, noch eine Situation, in der zwar viele verschiedene, dafür aber auf eher esoterisch anmutende Fehlertypen zurückgegriffen wird.

Anforderung 8.2

Das Rahmenwerk soll möglichst unterschiedliche Mutationen für verschiedenartige Fehlertypen unterstützen, so dass für ein konkretes Programm mit hoher Wahrscheinlichkeit Fehler vieler unterschiedlicher Arten implantiert werden können.

Für aussagekräftige, statistisch signifikante Resultate sind große Mengen fehlerhafter Programme besser geeignet als kleine Mengen, vgl. [And+06]. Daraus ergibt sich unmittelbar Anforderung 8.3. Was als ausreichend hinsichtlich der Menge an Mutanten anzusehen ist, kann allerdings nicht allgemein beantworten werden. Es gilt dabei zu bedenken, dass eine große Anzahl an Mutanten eine logistische Herausforderung bezüglich deren Verwaltung darstellen kann und mit einer längeren Laufzeit der Mutationsanalyse einhergeht.

Anforderung 8.3

Durch Art und Anzahl der Mutationsoperatoren soll das Rahmenwerk sicherstellen, dass eine ausreichend große Zahl an Mutanten generiert wird. Was als ausreichend

gilt, hängt von der Aufgabe (z. B. gewünschte statistische Signifikanz, Transformationseigenschaften etc.) ab.

Selbst eine kleine bis mittelgroße Anzahl von Mutanten erfordert bereits ein hohes Maß an Automatisierung, was Erzeugung, Verwaltung und Ausführung betrifft. Dieser Tatsäche trägt Anforderung 8.4 Rechnung.

Anforderung 8.4

Das Rahmenwerk soll einen hohen Grad an Automatisierung unterstützen. Hierzu zählen alle relevanten, wiederkehrenden Arbeitsschritte, wie beispielsweise die Erzeugung und Verwaltung der Mutanten sowie die Testausführung und -auswertung.

Da allerdings eine Automatisierung nur dann wirklich hilfreich ist, wenn sie auch den Bedürfnissen der Anwender gerecht wird, muss auch auf ein Mindestmaß an Konfigurierbarkeit geachtet werden, vgl. Anforderung 8.5.

Anforderung 8.5

Art, Anzahl und Anwendungsstelle der anzuwendenden Mutatoren sollten sich durch den Benutzer beeinflussen lassen.

Mit Anforderung 8.6 wird außerdem eingefordert, dass die Erzeugung der Mutanten ausreichend effizient erfolgen soll. Hierzu beschränken sich Mutationen typischerweise auf rein syntaktische Änderungen bzgl. des *SUT*, vgl. [DLS78].

Anforderung 8.6

Die automatisierte Erzeugung der Mutanten sollte möglichst effizient sein und sich auf syntaktische Änderungen beschränken.

Heuristiken zum Anschluss „nicht-überlebensfähiger“ Mutanten können allerdings dennoch den zusätzlichen Aufwand wert sein und sind nicht auszuschließen. Entsprechend erklärt sich auch die letzte Anforderung, Anforderung 8.7.

Anforderung 8.7

Die Semantik des SUT sollte, falls sinnvoll möglich, beachtet werden, so dass gültige (i. S. v. kompilierende) Mutanten entstehen.

8.3 Ansatz

Um *SDM*-Transformationen sinnvoll mutieren zu können, sind zwei Hauptaufgaben zu lösen. Einerseits müssen typische Fehler identifiziert werden, indem ein *Fehlermodelle* für *SDM*-Transformationen entwickelt wird. Im Gegensatz zu den bisher aufgetretenen Modellbegriffen, handelt es sich hierbei weder um ein formales Modell noch um eine Instanz eines Metamodells. Statt dessen handelt es sich um eine Vorstellung davon, was die typischen Fehlerursachen sind. Nach Binder ist ein Fehlermodell eine „Annahme darüber,

an welcher Stelle Fehler sehr wahrscheinlich auftreten und entdeckt werden können“, frei aus dem Englischen übersetzt, s. [Bin99, S. 51]. Außerdem liefert ein solches Modell eine Begründung dafür, warum bestimmte Aufwände zur Fehlersuche gerechtfertigt sind, vgl. ebenfalls [Bin99, S. 66]. Ein Fehlermodells muss dabei nicht grundsätzlich explizit sein, sondern kann auch rein als Idee bestehen. Eine explizite Beschreibung kann auf unterschiedliche Art und Weise erfolgen, beispielsweise informell als natürlichsprachlicher Text, strukturiert in Tabellenform oder als Taxonomie.

Neben dem Fehlermodell müssen andererseits auch die den *MT*- bzw. *GT*-spezifischen Fehlerarten entsprechenden *SDM-Mutationsoperatoren* entwickelt werden. Ihrer in Definition 5.23 festgelegten Aufgabe entsprechend, sorgen Sie dafür, einzelne Fehler der entsprechenden Art in eine *SDM*-Beschreibung einzubringen. Die Änderungen der Implementierung erfolgen dabei, wie auch bei klassischen Mutationsansätzen, auf der syntaktischen Ebene, vgl. z. B. [SMS09]. Allerdings muss dazu ggf. auch die Semantik beachtet werden, wie z. B. in [MBT06a] festgestellt. Den beiden Teilaspekten Fehlermodell und Mutationsoperatoren sind die nun folgenden Unterabschnitte 8.3.1 und 8.3.2 gewidmet.

8.3.1 Ein Fehlermodell für SDM-Transformationen

Der erste Schritt zur Entwicklung eines *SDM*-Fehlermodells liegt in der Bestandsaufnahme existierender Fehlermodelle anhand der einschlägigen Literatur. Anschließend wird das im weiteren Verlauf genutzte *SDM*-Fehlermodell vorgestellt.

Bestandsaufnahme existierender Fehlermodelle

OO-Ansätze Wie bereits in den Kapiteln 2 bis 4 dargestellt, weist die hier genutzte *MDSD*-Ausprägung aufgrund der Verwendung von Metamodellen auf Basis einfacher Klassendiagramme Ähnlichkeiten zu den weit verbreiteten *OO*-Ansätzen auf. Dieser Zusammenhang wird noch deutlicher, wenn, wie hier, generative Verfahren genutzt werden, an deren Ende ausführbarer Code einer *OO*-Programmiersprache steht. Insbesondere dann, wenn die Klassenhierarchie des Codes Parallelen zum Metamodell aufweist, was beispielsweise bei *EMF* der Fall ist. Typische Fehlerquellen für objektorientierte Programme für die Themenkomplexe *Vererbung*, *Polymorphie*, *Überladen* und *Überschreiben von Methoden* sowie das *Zusammenspiel zwischen Klassen* und, mit Abstrichen, *Sichtbarkeiten*, vgl. [KCM00; Che01] und auch [MOK05], existieren folglich auch im Kontext der hier zu testenden Transformationen. Allerdings tritt die Bedeutung dieser Fehlerquellen aufgrund der größeren Automatisierung und dem höheren Abstraktionsniveau im Falle von *MDSD* mit einer anderer Gewichtung auf, als bei den ursprünglichen *OO*-Ansätzen. Nichtsdestotrotz sollte ein umfassender Mutationstestansatz für Modelltransformationen, falls möglich, typische *OO*-Fehlerquellen und daraus abgeleitete Mutationsoperatoren berücksichtigen. Da hier im Speziellen allerdings nicht das mutationsbasierte Testen als solches im Vordergrund steht, sondern es primär um die Bewertung einer Testmenge mit engem Fokus auf den dynamischen Aspekten der Transformationen geht, soll hier auf eine dedizierte und eingehende Untersuchung der *OO*-Mutationsansätzen zugunsten besser angepasster Ansätze verzichtet werden.

UML, Klassendiagramme und Metamodelle Neben Arbeiten zu typischen *OO*-Fehlerquellen existieren einige Arbeiten, die sich mit dem Testen auf Basis von *UML*-

Modellen auseinander setzen. Neben reinen *MBT*-Ansätzen, vgl. Kapitel 5.5, bei denen Modelle als (fehlerfreie) Abstraktionen des zu testenden Systems gelten, behandeln andere Arbeiten das direkte Testen von UML-Modellen. Im Hinblick auf *fehlerbasiertes* Testen stehen hierbei typischerweise Fehler bei der Implementierung bzw. Abbildung von ausführbaren (Design-)Modellen auf Code im Mittelpunkt der Betrachtung, vgl. beispielsweise [Bin99, S. 569 ff] oder auch [OA99; AO00]. Bezogen auf das Ziel der Entwicklung eines Fehlermodells für *SDM*-Transformationen sind diese Arbeiten allerdings von eingeschränkter Relevanz, einerseits aufgrund der andersartigen Modellierung der Programmlogik (ausführbare UML-Modelle gegenüber programmierten Graphtransformationen), andererseits weil in dem hier zugrunde liegenden Szenario, so denn überhaupt Code generiert und kein interpretierter Ansatz genutzt wird, dieser aufgrund des Generatoreinsatzes von zumindest konstanter Qualität sein sollte. Arbeiten, die sich implizit auf Fehler und Inkonsistenzen in (UML-)Klassendiagrammen beziehen und diese entweder durch testende oder verifizierende Verfahren versuchen zu finden, wurden bereits in Kapitel 6.2 vorgestellt. Der unmittelbare Bezug dieser Arbeiten zur *statischen Semantik* der Metamodelle, welche den Transformationen zu Grunde liegen, ist unmittelbar erkennbar.

Allerdings lassen sich potentielle Fehlerquellen für Metamodelle aus den erwähnten Arbeiten entnehmen, ggf. ergänzt um angepasste Erkenntnisse. Als mögliche Fehlerquellen ergeben sich zum einen die gleichen Ursachen, wie sie auch bei den bereits angesprochenen *OO*-Ansätzen auftreten, zum anderen kristallisieren sich die folgenden Aspekte heraus: (i) Assoziationen und deren Multiplizitäten (insbesondere bei komplexeren Composite-Strukturen, Konsistenzanforderungen aufgrund von Bidirektionalität oder n-ären Assoziationen), (ii) Verfeinerungen[2] von Assoziationen (im Folgenden nicht weiter betrachtet), (iii) komplexwertige, abgeleitete Attribute oder komplexe Abhängigkeiten zwischen Attributen, (iv) OCL-Constraints.

Die Autoren von [Din+05] stellen darüber hinaus *explizit* ein einfaches Fehlermodell für *UML*-Designmodelle vor. Sie unterscheiden drei Kategorien möglicher Probleme:

1. Probleme, die sich in schlechten Werten bei Qualitätsmetriken äußern,
2. Fehler, die sich durch statische Analysen erkennen ließen, und die sich entweder
   (a) auf eine einzelne (globale) Sicht des Modells beziehen, oder
   (b) durch Inkonsistenzen zwischen unterschiedlichen Sichten ergeben,
3. Fehler, die sich bei einer dynamischen Ausführung manifestieren.

Darüber hinaus werden in der Arbeit auch einige Mutationsoperatoren für *UML*-Klassendiagramme vorgestellt und in die oben wiedergegebene Taxonomie einsortiert. Die vorgestellten Mutationen beziehen sich auf (i) Klassen (bzgl. Abstraktheit), (ii) Attribute (bzgl. Sichtbarkeit, Multiplizitäten), (iii) Methoden (bzgl. Sichtbarkeit und Parameterreihenfolgen, Redefinitionen löschen), (iv) Assoziationen (Listen- und Mengen-Semantiken, Navigierbarkeit, Multiplizitäten, Aggregationstyp abändern) oder (v) die Struktur (Vertauschen von Rollennamen kompatibler Assoziationsenden, Umbiegen von Assoziationen oder von Vererbungskanten).

Im Hinblick auf das Testen von *SDM*-Transformationen können in vielen Fällen die beteiligten Metamodelle als vorgegeben oder als unveränderlich angenommen werden, so dass zwar Fehler auf Ebene des Metamodells nicht ausgeschlossen sind, eine Berücksichtigung für das spezielle Vorhaben der Bewertung einer Testmenge für die Transformationen allerdings nicht unmittelbar zielführend erscheint, vgl. auch [SW08]. Allerdings soll

[2] Beispielsweise durch `subset`, `redefine` oder `union` Modifikatoren.

hierbei nicht unerwähnt bleiben, dass bei neu zu entwickelnden Modelltransformationen häufig auch die Metamodelle mit zu entwickeln sind. Darüber hinaus sind Änderungen an Metamodellen, z. B. aufgrund von Überlegungen im Rahmen der Umsetzung einer Transformation, nicht außergewöhnlich. Im Sinne eines allumfassenden Testkonzeptes auf Basis von Mutationen erscheint es also nicht ausgeschlossen, auch Metamodelle beim Testen zu mutieren.[3] Als wesentliche Erkenntnis für das hier verfolgte Vorhaben ist festzuhalten, dass sowohl das zu entwickelnde Fehlermodell als auch die eigentlich benötigten Mutationsoperatoren Aspekte Eigenheiten der Metamodelle berücksichtigen müssen.

Neben Klassendiagrammen bietet die *UML* mit den Aktivitätsdiagrammen noch einen weiteren Diagrammtyp mit Bezug zur Modellierung mit der *SDM*-Sprache, wie bereits in Abschnitt 4.3 beschrieben. Zur Erinnerung, der Kontrollfluss von *SDM*-Transformationen wird grafisch mit Hilfe einer an die konkrete Syntax von Aktivitätsdiagrammen angelehnten Sprache beschrieben. Zwar werden dabei nur die einfachen Kontrollflusskonzepte der Aktivitätsdiagramme verwendet und komplexere Konzepte, wie beispielsweise Parallelität, Datenflüsse, Signale, Ein- und Ausgabeparameter, komplexe Bedingungen etc. ausgeklammert. Dennoch sollen hier auch Arbeiten zur Mutation von Aktivitätsdiagrammen der Vollständigkeit halber erwähnt werden; das Testen solcher Diagramme ist bereits in Unterkapitel 6.2.2 thematisiert worden.

In [ZA06] beschreiben Zualkernan und Abu-Naaj die Architektur eines Webservices für die Mutationsanalyse von Aktivitätsdiagrammen. Sie schlagen vor, die sog. *HAZOP*-Methodik anzuwenden, um damit strukturiert und anhand sog. „Leitworte“ die Mutationsoperatoren abzuleiten. Ein konkretes Fehlermodell wird allerdings nicht deutlich herausgearbeitet. Dagegen überwiegen generische Beschreibungen der Auswirkungen verschiedener, durch die Methodik vorgegebener Situationen auf das beobachtbare Systemverhalten. Allerdings werden daraus insgesamt sechs konkrete Mutationsoperatoren abgeleitet, nämlich (i) das Hinzufügen von Kanten, (ii) das Hinzufügen von Aktionen, (iii) das Löschen von Kanten, (iv) das Löschen von Aktionen, (v) das Hinzufügen von Rückwärtskanten (inklusive dem Löschen von antiparallelen Kanten) sowie (vi) das Vertauschen von zwei Aktionen im Kontrollfluss.

In [FL08] stellen Farooq und Lam insgesamt 15 Mutationsoperatoren für den Kontrollflussteil von Aktivitätsdiagrammen vor. Auch werden vier Hauptfehlerquellen explizit identifiziert, nämlich Fehler bezüglich (i) der Reihenfolge von Aktionen, (ii) der Schnittstelle zur Umgebung, (iii) der Synchronisierung von Teilabläufen (die zu Verklemmungen oder Race-Conditions führen) sowie (iv) der Verzweigungungen des Kontrollflusses.

Modelltransformationen Mit den dynamischen Aspekten von Modellierung nähern wir uns dem eigentlichen Kern des zu lösenden Problems. Als primäre Quelle, in der die Anwendung der Mutationsmethodik im Sinne der Mutationsanalyse für das Testen von *MTs* erstmalig beschrieben wird, gilt gemeinhin [MBT06a]. In dieser Arbeit präsentieren Mottu et al. ein generisches Fehlermodell sowie eine Menge generischer Mutationsoperatoren für *MTs* im Allgemeinen und unabhängig von jeglicher Transformationssprache. Dazu werden vier abstrakte Operationen identifiziert, die typischerweise bei Implementierungen von Modelltransformationen zum Einsatz kommen, und die spezifische Fehlerarten aufweisen können. Konkret werden (i) das *Navigieren* entlang von Kanten innerhalb von

[3] Technisch ließe sich dies ggf. mit Werkzeugen wie dem „Ecore Mutator“ von Philip Langer umsetzen, der in [Tae+14] erwähnt wird. `https://code.google.com/a/eclipselabs.org/p/ecore-mutator/`

Modellen, (ii) das *Filtern* von Modellelementen, (iii) das *Schreiben* im Ausgabemodell bei exogenen Transformationen sowie (iv) das *Manipulieren* des Eingabemodell bei endogenen Transformationen genannt. Eine weitere Erkenntnis der Arbeit ist, dass klassische, *OO*-spezifische Mutationsoperatoren im *MT*-Kontext ungeeignet erscheinen, um sinnvoll wiederverwendet zu werden, da sie weder typische Fehler der Entwickler nachbilden noch zu Ergebnissen führen würden, die potentielle statische Überprüfungen oder das Kompilieren nicht fehlerfrei überstehen würden. Das bedeutet für den hier betrachteten Anwendungsfall, dass eine technisch mögliche Mutation auf Ebene des Java-Codes kein geeignetes Mittel darstellt! Dies wäre in etwa so, als würde man in eine Programmdarstellung auf Maschinencodeebene Fehler injiziert, um so ein zugrundeliegendes Hochsprachenprogramm zu testen oder entsprechende Tests für dieses zu bewerten; ein für das Mutationstesten höchst ungewöhnliches Vorgehen. Die ebenfalls von den Autoren aufgeführte fehlende Sprachunabhängigkeit der *OO*-Ansätze als Argument gegen klassische Mutationsoperatoren fiele hier aufgrund der gesetzten *SDM*-Sprache allerdings weniger ins Gewicht.

In [KH13] entwickeln Khan und Hassine auf Basis der generischen Mutationsoperatoren aus [MBT06a] entsprechende Adaptionen für *ATL*. Dabei sind allerdings von zehn vorgestellten Mutatoren nur vier auf Exemplare aus der zugrunde liegenden Arbeit zurückzuführen. Die restlichen Mutatoren sind neuartig und sprachspezifisch, beziehen sich also unmittelbar auf typische Fehler in *ATL*-Transformationen. Für *SDM*-Transformationen lassen sie sich daher nicht wiederverwenden.

In [KA07] identifizieren Küster und Abd-El-Razik sechs Fehlerarten in ihrem Fehlermodell für regelbasierte Modelltransformationen. Die Fehler beziehen sich dabei auf die folgenden Aspekte der Transformation, vgl. [KA07, Abschnitt 3.1]:

1. „Meta model coverage“ – Bei der Formulierung von Regeln können Elemente aus dem Metamodell übersehen werden, die so unberücksichtigt bleiben. Als Konsequenz werden korrekte Eingaben nicht bzw. nur unvollständig transformiert.
2. „Creation of syntactically incorrect models“ – Schreibende Operationen werden inkorrekt umgesetzt, wodurch sich Ausgabemodelle ergeben, die entweder nicht mehr Teil der Zielsprache sind oder Nebenbedingungen verletzen.
3. „Creation of semantically incorrect models“ – Vorbedingungen von Regeln sind fehlerhaft formuliert, so dass Regeln an den falschen Stelle angewendet werden können. Als Konsequenz kommt es im Fehlerfall zu einem syntaktisch korrekten aber semantisch fehlerhaften Ausgabemodell.
4. „Confluence“ – Die Konfluenzeigenschaft wird durch die Regelmenge verletzt. Bei mehrfacher bzw. wiederholter Anwendung von Regeln oder Regelsequenzen können unterschiedliche Ausgaben entstehen. Auch sind Zwischenmodelle denkbar, die nicht weiter transformierbar sind.
5. „Correctness of transformation semantics“ – Eine gewünschte oder vorausgesetzte Eigenschaft der Transformation syntaktischer, semantischer oder sonstiger Natur ist nicht gegeben oder wird verletzt. Als Beispiel wird die fehlende Verklemmungsfreiheit einer Transformation genannt.
6. „Errors due to incorrect coding“ – Hierunter werden sonstige Fehler ohne direkten Bezug zu den zuvor genannten Punkten zusammengefasst.

Im Text werden darüber hinaus noch zwei weitere fehleranfällige Punkte deutlich, nämlich das Zusammenspiel von Regeln – problematisch sind hierbei beispielsweise unbeabsichtigte oder übersehene Konflikte oder Abhängigkeiten – sowie das Einhalten zusätzlicher

Integritäts- und Konsistenzbedingungen, ausgedrückt z. B. durch *OCL*-Constraints.

In [DPV08] stellen Darabos et al. ein Fehlermodell für das Pattern-Matching (insb. auch für die Implementierung dessen) bei regelbasierten Graphersetzungen vor. Die Fehlerarten verteilen sich, den Autoren nach, auf zwei Unterkategorien, nämlich (a) Grundsätzliche Implementierungsfehler bei *MTs* sowie (b) Fehler, welche spezifisch für die Mustersuche bei Graphtransformationen sind. Zu den Implementierungsfehlern zählen den Autoren zufolge (i) Fehler durch *Auslassungen*, z. B. weil obligatorische Elemente des Musters fehlen, (ii) Fehler durch *Vertauschen*, z. B. weil der falsche Typ für einen der Knoten des Musters gewählt wurde sowie (iii) Fehler durch *Seiteneffekte* aufgrund von redundanten/überflüssigen Elementen. Zu den Graphtransformationsfehlern zählen entsprechend (i) Fehler, die zur Entstehung von Dangling-Edges, vgl. Abschnitt 4.2.1, führen und (ii) Fehler, die ein (fehlerhaftes) nicht-injektives Abbilden von Elementen aus dem Muster auf Elemente aus dem Modell betreffen.

Weitere Hinweise auf potentielle und allgemeine Fehlerquellen bei der Formulierung von *MTs* liefert die Taxonomie von Wimmer et al. aus [Wim+09, s. Abb. 5]. Darin werden Fehler anhand ihres Auftrittskontextes unterschieden, nämlich in (i) Metamodell-, (ii) Modell- oder (iii) Transformationslogikfehler. Wird bei den ersten beiden Klassen nur grob zwischen syntaktische und semantische Fehler unterschieden, ist die dritte Klasse feiner untergliedert. Beispielsweise wird noch mal zwischen Fehlern innerhalb einer Regel und Fehlern, die durch das Zusammenspiel von Regeln entstehen, unterschieden.

Arbeiten wie [MBT06a; KA07; Wim+09], die Fehler in *MTs* in allgemeiner Form klassifizieren, können als Orientierung bei der Entwicklung von Fehlermodellen für konkrete Sprachen, wie auf den folgenden Seiten, dienen. Sie haben auch im Laufe dieser Arbeit dabei geholfen den Blick für die im folgenden beschriebenen konkreten Fehlerarten zu schärfen. Denn grundsätzlich können konkrete Fehlermodelle als Instanziierung generischer Fehlermodell aufgefasst werden.

Das SDM-Fehlermodell

Nach der Beschreibung existierender Arbeiten wird im Folgenden das hier entwickelte, konkrete Fehlermodell für die *SDM*-Sprache vorgestellt. Es dient im Anschluss daran als Motivation für die als sinnvoll erachteten und beschriebenen Mutationsoperatoren. Da die *SDM*-Sprache im Wesentlichen aus zwei Teilsprachen besteht, werden die Fehlerklassen für beide Teilsprachen separat betrachtet. Einige der Fehlerarten lassen sich bereits durch statische Analysen oder aufgrund von Widersprüchen bei der Codegenerierung erkennen; sie werden im Folgenden durch das ‘‡’-Zeichen gekennzeichnet und sind für das dynamische Testen der Transformation uninteressant. Entsprechende Fälle können allerdings für das Testen der Werkzeugkette Relevanz besitzen.

SDM-Kontrollflussgraph Ein Teil der möglichen Fehler bei der Verwendung von *SDM*-Transformationen betrifft den Kontrollfluss, mit dem bekanntermaßen Regeln ausgewählt und deren Anwendung gesteuert wird. Wie bereits in Kapitel 4.3.1 deutlich geworden sein sollte, repräsentiert der Kontrollfluss eine Art *Control Flow Graph (CFG)*.

Wohlgeformtheit‡ Eine erste Fehlerkategorie ergibt sich durch Verletzungen der *Wohlgeformtheit* des Kontrollflusses. Fehler dieser Art sind in der Regel syntaktischer Natur und lassen sich mit Hilfe statischer Analysen entdecken. Allerdings ist das Metamodell,

s. Anhang E, Abbildung E.2, an einigen Stellen nicht restriktiv genug, so dass zusätzliche Einschränkungen anderweitig formuliert werden müssen. Beispielsweise ist es nicht grundsätzlich ausgeschlossen, dass auch ein Startknoten eingehende oder ein Stoppknoten ausgehende Kontrollflusskanten aufweist. Die notwendigen Einschränkungen für ein syntaktisch und semantisch valides Modell liegen somit nicht für alle Fälle explizit vor. Folglich legt de facto der verwendete *SDM*-Editor im Zusammenspiel mit dem Codegenerator fest, was zulässige Modellinstanzen sind und was nicht. Leider waren und sind diese stetigen Änderungen und Anpassungen unterworfen, so dass die Gefahr fehlerhafter Spezifikationen hinsichtlich des Kontrollflusses nicht vollständig auszuräumen ist. Bezüglich der Wohlgeformtheit der Kontrollflusses können sich – unter Vernachlässigung statischer Analysen – die folgenden Fehler ergeben.

Unerreichbare Knoten‡ Der *CFG* enthält Knoten, welche keine Startknoten sind und die keine eingehenden Kanten aufweisen, also die nachfolgende Invariante verletzt ist.

```
context ActivityNode
  inv "no non-reachable nodes":
    (not self.oclIsTypeOf(StartNode)) implies (self.incoming->size() > 0)
```

Zu dieser Art des Fehlers kann es beispielsweise dann leicht kommen, wenn beim „Umhängen“ bestehender Kanten vergessen wird, eine neue eingehende Kante zu ziehen, oder wenn zwei direkt verbundene Knoten einzeln und nacheinander (in Richtung des Kontrollflusses) gelöscht werden sollen, wobei nur der erste Knoten tatsächlich entfernt wird (inklusive aller angrenzenden Kanten), der zweite aber versehentlich nur aus der Visualisierung des Diagramms entfernt wird.

Sackgassen‡ Der *CFG* enthält Knoten, die nicht vom Typ `StopNode` sind und die keine ausgehende Kante aufweisen. Erreicht der Kontrollfluss einen solchen Knoten, könnte dieser nicht mehr verlassen werden; die Ausführung müsste stoppen. Im Falle eines solchen Fehlers wird die nachfolgende Bedingung verletzt.

```
context ActivityNode
  inv "no dead ends":
    (not self.oclIsTypeOf(StopNode)) implies (self.outgoing->size() > 0)
```

Unstrukturierter CFG‡ Abgesehen von den bisher erwähnten, eher offensichtlichen Einschränkungen der grundlegenden Struktur des *CFG* existiert eine weitere wichtige Einschränkung: der Kontrollfluss einer *SDM*-Implementierung sollte im weitesten Sinne[4] den Anforderungen der strukturierten Programmierung genügen. D. h. es soll ausgeschlossen sein, dass der Kontrollfluss so spezifiziert wird, dass er isomorph zu dem eines *Goto*-Programms ist, welches nicht den Anforderungen der strukturierten Programmierung genügt. Für diese Forderung sprechen zwei wichtige Gründe, nämlich, dass erstens die Goto-Programmierung gemeinhin als schlechter Stil – insbesondere aufgrund der schlechten Les- und Wartbarkeit – angesehen wird, vgl. [Dij68], und dass zweitens eine direkte und Goto-freie Abbildung auf Java-Code gegeben sein sollte. Letztere ist aus theoretischer Sicht zwar immer möglich, wie in

[4] Mehrere Stoppknoten sind hier explizit erlaubt.

[BJ66] gezeigt wurde, allerdings gestaltet sich die Abbildung leichter, wenn bereits der *CFG* der *SDM*-Implementierung eine entsprechende Form aufweist. Darüber hinaus stellt auf der technischen Ebene der Codegenerator des eingesetzten Graphtransformationswerkzeugs eMoflon, der auf der *CodeGen2*-Komponente von Fujaba [NNZ00] basiert, entsprechende Anforderungen an die *CFG*-Struktur der Eingabe.

Formal lässt sich die Strukturiertheit-Eigenschaft mit Hilfe der Untersuchung der sog. *Reduzierbarkeit* (von engl. *reducible*) bzw. *Faltbarkeit* (von engl. *collapsible*) nachweisen, vgl. [All70; HU72] und auch [HU74]. Bei *SDM*-Diagrammen wäre dabei allerdings noch zwischen „normalen“ Pfaden, die keine Each-Time-Kante traversieren, und Each-Time-Zyklen zu unterscheiden, vgl. 4.3.1.

In der genutzten Werkzeugversion ist es relativ schnell geschehen, dass ein fehlerhafter Kontrollfluss spezifiziert ist, insbesondere bei unerfahrenen Entwicklern. Begünstigt werden entsprechende Fehler durch die zweidimensionalen grafischen Diagramme mit der einhergehenden freien Platzierbarkeit von Knoten und Kanten. Insbesondere bei Änderungen an bestehenden Transformationen können sich entsprechende Fehler leicht einschleichen.

Each-Time-Schleife falsch verlassen‡ Falls mindestens ein Pfad von einem Each-Time-Knoten aus über die zugehörige Each-Time-Kante zu einem Knoten existiert, der nicht Teil der Each-Time-Komponente ist, und bei dem nicht zuvor der Each-Time-Knoten wieder erreicht und über die zugehörige „End“-Kante verlassen wurde, liegt ein Fehler vor. In dem Fall würde die Each-Time-Schleife auf fehlerhafte Weise verlassen worden sein. Da der Codegenerator in solchen Situationen allerdings einen Fehler meldet, tritt dieser Fehler in der Praxis nicht auf.

Each-Time-Schleife falsch betreten‡ Als komplementär zum vorherigen Fall kann grundsätzlich der Fall angesehen werden, dass ein Pfad zu einem Knoten innerhalb der Each-Time-Schleife existiert, der nicht zuvor den zugehörigen Each-Time-Knoten der Komponente passiert. Dies entspricht einem fehlerhaften Betreten der Schleife. Auch in diesem Fall würde der Codegenerator den Fehler in der Spezifikation erkennen.

Logikfehler Die Klasse der Fehler zur Programmlogik des Kontrollflusses umfasst sowohl Fehler semantischer als auch syntaktischer Art. Es ergeben sich beispielsweise Fehler hinsichtlich einer falschen Verzweigung aufgrund fehlerhafter Verzweigungsbedingungen. Im Folgenden sind die wesentlichen Fälle aufgelistet.

Vertauschen von Bedingungen Soll der Kontrollfluss anhand des Ergebnisses der Regelauswertung oder eines Methodenaufrufs verzweigt werden, müssen zwei ausgehende Transitionen vorhanden sein: eine mit der Verzweigungsbedingung `[Success]` und eine mit der Bedingung `[Failure]`. Dies lässt sich leicht durch eine syntaktische Überprüfungen sicherstellen. Allerdings ist es grundsätzlich auch möglich, dass beide Fälle verwechselt und unbeabsichtigt vertauschen werden, was in einem semantischen Fehler resultiert. Fehler dieser Art lassen sich nur eingeschränkt mittels statischer Analysen abfangen.

Eine vergleichbare Situation existiert auch für Each-Time-Knoten. Hier tragen die beiden ausgehenden Kanten, so denn beide vorhanden sind, die Verzweigungsbedingungen „[End]“ und „[Each Time]“ (letztere Kante ist optional). Auch diese beiden Kanten können vertauscht werden. Allerdings ist aufgrund der restriktiven Einschränkungen für einen validen Kontrollfluss selten mit einem kompilierenden Endergebnis für einen solchen Fehlerfall zu rechnen.

Inkonsistente Kombinationen von `EdgeGuard`-Werten‡ Nicht anhand des Metamodells E.2 zu erkennen, sind dennoch nur bestimmte Kombinationen von `EdgeGuard`-Literalen bei zwei ausgehenden Kanten valide. So ist der Wächter-Wert `SUCCESS` nur in Kombination mit vorhandenem `FAILURE`-Wert zulässig und der Wert `EACH_TIME` kann nur zusammen mit `END` auftreten. Entsprechende Überprüfungen sind als statische Analyse leicht zu implementieren und im hier eingesetzten *SDM*-Editor ebenfalls umgesetzt.

Eine Sonderstellung aufgrund des relativ häufigen Auftretens nimmt der Fall ein, in dem „[Success]“- oder „[Failure]“-Kanten fälschlicher weise an einem For-Each-Knoten entspringen. Dieser Fehler entsteht beispielsweise, wenn ein bestehender Story-Knoten in einen For-Each-Knoten umgewandelt wird, anschließend aber vergessen wird, vorhandene ausgehende Kanten anzupassen.

Redundanzfehler Die Klasse der Redundanzfehler für den Kontrollfluss fasst Fehler zusammen, die durch überflüssige und zum Teil widersprüchliche Elemente im Diagramm entstehen. Die nachfolgend aufgelisteten Fälle sind denkbar.

Zu viele (falsche) ausgehende Kanten‡ Ein Startknoten mit mehr als einer ausgehende Kante oder aber nur einer Kante, die dafür aber eine Bedingung aufweist, sind zwei offenkundige Fehlerfälle, bei denen die Spezifikation zu viele Elemente bzw. Angaben umfasst. Auch sind bei Story-Knoten nie mehr als zwei ausgehende Kanten sinnvoll. Selbst wenn eine Success-, eine Failure- und eine Kante ohne Bedingung kombiniert würden, so wäre dies im besten Fall nur sinnlose Redundanz. Falls solch ein Kantentripel dabei nicht auf den gleichen Nachfolger zeigen würde, hätte man bereits einen Widerspruch.

Mehrere Startknoten‡ Eine Operationsbeschreibung hat per Definition nur einen Startknoten. Das Metamodell schließt allerdings mehrere Startknoten nicht aus und so ist es von statischen Analysen und der Implementierung der Codegenerierung abhängig, wie mit solch einer Situation tatsächlich umgegangen wird. Sind mehrere Startknoten vorhanden, so handelt es sich in jedem Fall um eine fehlerhafte Spezifikation. Eine entsprechende statische Analyse ist unmittelbar und leicht zu implementieren, so dass dieser Fehlertyp keine praktische Bedeutung besitzt.

Überflüssige Fallunterscheidungen‡ Es kann vorkommen, dass eine Regel in einem Story-Knoten aufgrund des Inhalts nie fehlschlagen kann, so dass eine Failure-Kante niemals traversiert werden würde. Beispiele hierfür sind (i) der Story-Knoten, in dem ausschließlich die `this`-Variable durch Wiederholung eingeführt wird, (ii) Regeln, in denen ausschließlich Attribute eines bereits gebundenen Knotens atomar manipuliert werden sowie (iii) Regeln, deren *LHS* leer ist, und bei denen nur Elemente im Modell neu angelegt werden. Bei den beiden letzteren Fällen können zwar

theoretisch Laufzeitfehler auftreten, diese sind allerdings in der hier eingesetzten *SDM*-Sprachversion nicht auf Ebene der *SDM*-Implementierung abzufangen. In allen genannten Fällen ist eine Fallunterscheidung mittels Success- und Failure-Kante mindestens überflüssig, wenn nicht sogar irreführend. Es handelt sich um eine potentielle Fehlerquelle. Eine statische Analyse ist nicht ausgeschlossen, aber tendenziell aufwendiger zu implementieren.

Auslassungsfehler Bei Auslassungsfehlern bezüglich des Kontrollflusses geht es um die fehlende Berücksichtigung eigentlich zu behandelnder Fälle. Insbesondere besteht die Gefahr fehlender Verzweigungen und damit auch fehlender Pfade.

Fehlende Kante Ein sehr häufiger Fehler liegt darin, dass ein Entwickler beim Spezifizieren des Kontrollflusses nicht bedenkt, dass eine bestimmte Musterauswertung auch fehlschlagen kann. Als Konsequenz entsteht dann entweder eine Situation, in der nur eine ausgehende Success-Kante spezifiziert wird, was durch eine entsprechende statische Analyse leicht abgefangen werden kann, oder eine Situation, in der nur eine Kante ohne Bedingung spezifiziert wurde. Je nach Inhalt und Bedeutung der Regel sollte aber ein Fehlschlagen gesondert behandelt werden, was in letzterem Fall fälschlicher Weise nicht geschehen würde. Eine statische Analyse könnte diesen Fall nicht sicher erkennen, ggf. aber eine Warnung aussprechen.

Pfad fehlt Der vorangegangene Fehler kann ein Indiz dafür sein, dass ein ganzer Pfad im Kontrollflussgraph fehlt. Beispielsweise ist es denkbar, dass eine komplette Each-Time-Schleife für einen For-Each-Knoten vergessen wurde.

Fehlende Bedingung‡ Ein häufig vorkommender Flüchtigkeitsfehler liegt darin, dass vergessen wird, den noch fehlenden Wächter einer Transition anzugeben. In der Praxis ist dies allerdings sehr einfach mit Hilfe einer statischen Analyse zu entdecken, so dass dieser Fehler bei Beschreibungen, die mittels Editor erstellt wurden, nicht auftritt.

Fehlerhaftes Zusammenspiel zwischen Kontrollfluss und Graphersetzungsregeln Da der Kontrollfluss die Anwendung der Regeln steuert, sind Fehler im Zusammenspiel beider Teilsprachen möglich. Tatsächlich existieren im Zusammenspiel beider Sprachen Fallstricke, die zu fehlerhaften Spezifikationen führen können. Im Folgenden werden drei Konstellationen betrachtet.

Variablen-Geltungsbereich fehlerhaft‡ Die Werte gebundener *OVs* in den Mustern ergeben sich zum Großteil erst durch die Auswertung anderer Regeln, die im Ablauf früher ausgewertet werden. Schlägt die Auswertung einer Regel, in der eine *OV* an einen Knoten aus dem Modell gebunden werden soll, allerdings fehl, so ist davon auszugehen, dass die entsprechende Variable nicht sinnvoll initialisiert wurde. Wurde die Variable allerdings bereits an anderer Stelle an einen Wert gebunden, kann es zur Laufzeit so aussehen, als wäre die Variable tatsächlich sinnvoll belegt. Bisher fehlt in der hier verwendeten *SDM*-Sprache ein sogenanntes *Scope*-Konzept für *OVs*, das den Gültigkeitsbereich für Variablen klar definiert und analysierbar macht.[5] Somit sind diesbezügliche Fehler nicht ausgeschlossen.

[5] Während der Erstellung dieser Arbeit wird an statischen Analysen gearbeitet, die solche problematischen Situationen aufdecken sollen.

Sequenz statt großer Regel Manchmal kann es vorkommen, dass die Suche nach einem bestimmten Teilgraphen auf unterschiedliche Weise erfolgen kann. So kann es beispielsweise möglich sein, ein relativ großes und komplexes Muster zu spezifizieren. Oder aber es wird eine Menge kleinerer Teilmuster sequenziell hintereinander abgearbeitet. Unter gewissen Umständen sind beide Varianten dahingehend äquivalent, dass sie zum selben Endergebnis führen. Allerdings kann es sein, dass die Verwendung einer Sequenz von Regeln dazu führt, dass nicht alle möglichen Permutationen beim Binden durchprobiert werden, so dass ein eigentlich vorhandener Treffer nicht gefunden wird. Entsprechende Fehler können schwer zu finden sein. Grundsätzlich erscheinen auch Implementierungen mit möglichst wenigen und möglichst umfangreichen Mustern erstrebenswerter zu sein, da sie den deklarativen Teil der Sprache besser ausnutzen und dadurch beispielsweise die Lesbarkeit aufgrund kompakterer Spezifikationen und besserer Nachvollziehbarkeit deutlich verbessern.

Umfangreiche Regel statt Sequenz Leider ist auch die angestrebte Lösung mit wenigen Mustern nicht vor Fehlern gefeit, wenn nämlich versucht wird, zu viele Dinge in einer einzigen Regel zu erreichen. Kann beispielsweise ein Teil des komplexen Musters nicht gefunden werden, schlägt die komplette Regelauswertung im Standardfall fehl, was ggf. zu „grobkörnig“ ist, da Teilschritte dennoch auszuführen wären. Eine Aufteilung in mehrere kleinere Regeln mit entsprechender Verknüpfung über den Kontrollfluss wäre dann der eigentlich richtige Weg.

SDM-Knoten Neben den Kontrollflussfehlern bezieht sich eine weitere große Klasse an Fehlern auf den Inhalt und die Eigenschaften einzelner Knoten in einem *SDM*-Programm. Insbesondere der Inhalt von Story-Knoten steht hierbei im Fokus der Betrachtung.

ActivityNode Als abstrakte Basisklasse aller Knoten in *SDM*-Diagrammen sind für `ActivityNode`s Referenzen auf eingehende und ausgehende Kontrollflusskanten definiert. Zusätzlich erbt die Klasse die beiden `EString`-Attribute `name` und `comment`. Fehler bezüglich der Kontrollflusskanten wurden bereits eingehend betrachtet. Die optional möglichen Kommentare sollen hier vernachlässigt werden, da sie in der Praxis äußerst selten genutzt werden. Bleibt folglich noch der Name eines Knotens als Quelle möglicher Fehler.

(Nicht-)leerer Name‡ Für die meisten konkreten Unterklassen ist die Benennung nicht explizit notwendig. Start- und Stoppknoten benötigen zum Beispiel keine Bezeichnung, somit sollte der Wert des `name`-Attributs für entsprechende Instanzen auch der leeren String sein. Dagegen tragen `StoryNode`- sowie `StatementNode`-Instanzen in der Regel einen Namen. Häufig wird dieser von Entwicklern dazu benutzt, die Bedeutung des Knotens hervorzuheben oder dessen Zweck rudimentär zu dokumentieren. Dementsprechend kann der Name auch als Kommentar im Generat auftauchen, was im Rahmen der verwendeten Codegenerierungstemplates tatsächlich auch der Fall ist. Somit sollte der Name für diese Knoten vorzugsweise nicht leer sein. Verschärft wird diese Forderung noch dadurch, dass beim *RPC*-Ansatz in der aktuellen Implementierung der Name als eindeutiger Bezeichner angesehen wird. Folglich darf der Name – neben weiteren Einschränkungen – *nicht* leer sein.

Fehlerhafter Name‡ Wie bereits angedeutet, werden Namen von `StoryNode`s bzw. `StatementNode`s dazu genutzt, um Kommentare im (Java-)Code zu erzeugen. Folglich

sollten die Namen keine Steuerzeichen enthalten, die den Compiler bzw. den Parser verwirren würden. Auch aufgrund des bei einer Persistierung respektive Serialisierung von *SDM*-Transformationen eingesetzten *XMI*/*XML*s sind bestimmte Sonderzeichen auszuschließen. Sind diese Bedingungen verletzt, ist von einer fehlerhaften Spezifikation auszugehen, wobei sich entsprechende Fehler durch einfache String-Überprüfungen leicht ausschließen lassen.

Nicht-eindeutiger Name‡ Werden Namen vergeben, so sollten diese innerhalb eines Diagramms grundsätzlich eindeutig sein, auch um die Identifikation eines Knotens zu erleichtern. Im Falle des *RPC*-Ansatzes wird diese grundsätzliche Empfehlung wiederum zu einer Notwendigkeit, um die Eindeutigkeit abgeleiteter Operations- und Methodennamen sicherzustellen. Folglich sind nicht-eindeutige Namen hier ebenfalls als Fehler anzusehen.

StartNode‡ Ein Startknoten weist keine innere Struktur auf. Dadurch ist es auch nicht möglich, bei seiner Spezifikation diesbezügliche Fehler in die Implementierung einzubringen; der Fall mehrerer Startknoten wurde bereits im Kontrollflussteil des Fehlermodells betrachtet. Der Knotentyp ist hier nur der Vollständigkeit halber aufgeführt. Allerdings kann, je nach eingesetztem Editor, der Startknoten eine manuell anpassbare Beschriftung tragen, die grundsätzlich immer der textuellen Repräsentation der Operation bzw. deren Signatur entsprechen sollte. Nur so ist sichergestellt, dass beispielsweise mehrere typkompatible Operationsparameter anhand ihres Namens sicher unterschieden werden können. Eine statische Analyse des Editors schließt Fehler hinsichtlich solcher Inkonsistenzen sicher aus.

StopNode Die Fehlerquelle bei Stoppknoten liegt in der zugehörigen Angabe der Rückgabewerte als Ergebnis der Operationsauswertung oder aber auch bei Aufrufen anderer Operationen (vergleichbar zur Situation bei `StatementNode`s). Grundsätzlich sind die möglichen Fehlertypen vergleichbar zu Fehlertypen bei Rückgabewerten von Methoden in allgemeinen *OO*-Programmen. Allerdings ist die statische Typüberprüfung bei den hier eingesetzten *SDM*-Werkzeugen noch nicht so ausgereift, so dass Fehler verhältnismäßig leicht unentdeckt bleiben können.

Falscher Rückgabewert Wird statt dem eigentlich erwartetem Rückgabewert ein falscher Wert mit korrektem Typ zurückgeliefert, so liegt dieser Fehlertyp vor. Eine mögliche Ursache kann darin liegen, dass der Wert einer inkorrekten *OV* oder ein falsches Attribut zurückgeliefert wird.

Falscher Rückgabetyp‡ Bei dieser Art von Fehler ist schon der Rückgabe*typ* eines Stoppknotens falsch. Mit einer Typanalyse ließe sich dieser Fehler zur Kompilierzeit statisch entdecken. Spätestens beim Kompilieren des generierten (Java-)Zielsprachenprogramms fällt dieser Fehlertypus aber auch hier auf. In der Praxis sind Fehler dieser Art kein echtes Problem.

Fehlende Rückgabe‡ Angenommen, die betrachtete Operation soll als Ergebnis einen Rückgabewert liefern, aber ein entsprechender Stoppknoten umfasst keine Definition eines Rückgabewertes, dann liegt ein Fehler dieser Art vor. Eine statische Überprüfung kann solche problematischen Situationen relativ leicht identifizieren. Einzig

bei der Verwendung von frei formulierbaren `LiteralExpression`s ist eine Analyse schwierig bis unmöglich, da hierzu beliebiger Code analysiert werden müsste.

Rückgabe bei rückgabeloser Operation‡ Definiert ein Stoppknoten einen Rückgabewert für eine Operation, die keinen Rückgabetypen umfasst, die also vom Typ `void` ist, so liegt offensichtlich ein weiterer Fehlerfall vor. Grundsätzlich könnte die Codegenerierung eine solche Situation mehr oder weniger stillschweigend „wegoptimieren“ bzw. ignorieren. Letztendlich handelt es sich aber um einen Fehlerfall, der ggf. auf tiefer liegende Missverständnisse hinsichtlich der Funktionsweise der Operation hindeuten kann und entsprechend gemeldet und korrigiert werden sollte.

Expression fehlerhaft Bei der Formulierung einer Anweisung zur Bestimmung des Rückgabewertes kann es zu den unterschiedlichsten Problemen kommen, jeweils in Abhängigkeit der Art der Anweisung. Die Details sind weiter unten in einem eigenen Abschnitt zu Fehlern bei `Expression`s beschrieben. Fehler dieser Art lassen sich teilweise durch statische Analysen entdecken.

StatementNode Mit Hilfe von `StatementNode`s können im Metamodell definierte (Hilfs-)Operationen mittels `MethodCallExpression` aufgerufen werden und gegebenenfalls der Kontrollfluss anhand des Rückgabewertes verzweigt werden. Andere `Expression`-Arten sind hierbei allerdings nicht effektiv ausgeschlossen, wobei die Interpretation in diesen Fälle unklar bzw. undefiniert ist. Typische Fehler bezüglich der `StatementNode`-Instanzen ergeben sich folglich dann, wenn entweder (a) keine `Expression` definiert wurde oder (b) eine spezifizierte Anweisung inhärent fehlerhaft ist, vgl. hierzu den entsprechenden Abschnitt weiter unten zu den `Expression`s.

StoryNode Potentielle Fehlertypen der Story-Konten bilden den wichtigsten Teil des Fehlermodells, da hierin Fehler des Graphtransformationsteil der *SDM*-Sprache mit enthalten sind. Betrachtet man die innere Struktur eines Story-Knotens, so sind zwei fehleranfällige Teilaspekte zu unterscheiden, nämlich (a) die Struktur der Regel, insbesondere im Hinblick auf die Suche nach den Mustern im Modell sowie (b) die Spezifikation der eigentlichen Ersetzungsschritte und den damit verbundenen, schreibenden Operationsschritten, die auf einen Treffer angewendet werden.

Bezogen auf Teilaspekt (a) lassen sich Fehlerarten dahingehend klassifizieren, bei welcher Teilaufgabe sie typischerweise entstehen oder worin ihre Hauptauswirkung besteht. Folgende konkrete Klassen werden hier unterschieden.

Variablendefinition Regeln werden in Form von *OVs* und *LVs* angegeben. Da Variablen durch Metamodellelemente getypt sind, ist es möglich, sich bei der Auswahl des jeweiligen Typs der Variable zu irren und einen ungewollten Typ auszuwählen. Darüber hinaus muss beim Anlegen einer Variable festgelegt werden, welchen `BindingOperator` (s. S. 63) und welche `BindingSemantics` (s. S. 64) sie tragen soll, vgl. nochmals Abbildung 4.4. Hierbei ist es ebenfalls nicht ausgeschlossen, dass entsprechende Flüchtigkeitsfehler auftreten. Allerdings fallen Fehler dieser Art potentiell auch durch Inkonsistenzen innerhalb der Regel auf oder die Auswirkungen auf das resultierende Verhalten sind so gravierend, dass die Fehler bereits bei einem unstrukturierten Testen auffallen.

Regelumfang Von Fehlern mit Bezug zum Regelumfang ist dann die Rede, wenn eine Regel entweder Elemente – also *LVs* oder *OVs* – umfasst, die überflüssig bzw. fehl am Platze sind oder Elemente fehlen, die eigentlich vorhanden sein müssten, um eine angestrebte Aufgabe zu lösen.

Spezifische OV-Fehler Bei *SDM*-Transformationen weisen *OVs* mit ihren Binding-Zuständen sowie ihren Attributen mehr Eigenschaften auf, auf die sich Fehler beziehen können, als *LVs*. So kann z. B. schon bei der Definition einer *OV* ein falscher `BindingState` vorgeben werden, was entweder dazu führen kann, dass ein unerwartetes Objekt an die Variable gebunden ist oder eine eigentlich gewünschte Belegung versehentlich aufgelöst wird.

Auch bietet das Sprachmittel der `BindingExpression` leider viel Raum, um eine Transformation fälschlicherweise so zu spezifizieren, dass entsprechende Fehler erst zur Laufzeit zu Tage treten. So kann die `BindingExpression`, wie andere `Expression`s auch, unmittelbar selbst fehlerhaft formuliert sein. Darüber hinaus ist es möglich, dass die Zuweisung durch eine implizite Typumwandlung fehlschlägt. Entweder, weil tatsächlich die Typen inkompatibel sind oder weil versucht wird, der Variablen einen ungültigen Wert, sprich `null`, zuzuweisen.

Eine weitere Quelle für Fehler betrifft Attributbedingungen von *OVs*. Sind `AttributeValueExpression`s von Variablen fehlerhaft, fehlen sie ganz oder sind zu viele Bedingungen formuliert, hat dies unmittelbaren Einfluss auf die Menge passender Treffer im Modell.

Regelsemantik und Auswertungsfehler Fehler, die sich keiner der zuvor aufgeführten Kategorien zuordnen lassen, betreffen i. d. R. die Auswertung bzw. Interpretation der Regeln. So kann es beispielsweise einen Fehler darstellen, wenn mehrere gebundene Variablen innerhalb einer Regel an das selbe Objekt im Modell gebunden sind. Das ist durchaus möglich, falls sich die vorangegangenen Bindungsvorgänge auf unterschiedliche Muster verteilen. In einem solchen Fall kann es dann dazu kommen, dass ein vordergründig vollständiger Match keinen echten Treffer darstellt, was möglicherweise erst durch gründliches Testen entdeckt werden kann.

Weitere problematische Situationen sind einerseits inkonsistente Regelanteile, bei denen sich beispielsweise anzulegende und zu löschende Elemente widersprechen, z. B. indem ein neu anzulegender Knoten mit einem zu löschenden Knoten neu verbunden werden soll – was grundsätzlich auch statisch überprüfbar ist – sowie Regeln, deren Musterteil zu keinem validen Suchplan führt. Letzterer Punkt kann allerdings nicht durch dynamisches Testen überprüft werden, da zur Erzeugung der ausführbaren Artefakte während der Codegenerierung bereits der Suchplan benötigt wird. Dieser Fehlerfall ist folglich nur der Vollständigkeit halber aufgeführt.

Bezogen auf Teilaspekt (b) ergeben sich Fehler durch unbeabsichtigte, falsche oder ausbleibende Modifikationen eines passenden oder mehrerer passender Teilgraphen. Genauer aufgeschlüsselt kann es zu den folgenden unerwünschten Situationen kommen:

Erzeugende Anteile fehlerhaft Die Ausgangssituation ist, dass neue Elemente im Modellgraphen angelegt werden sollen. Dabei kann es allerdings passieren, dass die neuen Elemente nur *unvollständig*, also fehlerhaft oder überhaupt nicht, angelegt werden.

So können eigentlich anzulegende Knoten oder Kanten in der Regel vergessen worden sein. Bei Kanten können einzelne Enden falsch ausgewählt werden. Bei *OVs* sind dagegen fehlende Attributbelegungen denkbar. Darüber hinaus ist es möglich, dass semantisch falsche Elemente – beispielsweise durch Wahl des falschen Typs – im Modell erzeugt werden.

Modifizierende Anteile fehlerhaft Sollen bestehende Elemente des Modells nur modifiziert werden, so betrifft dies in dem Kontext dieser Arbeit ausschließlich den Fall, dass *OVs* (und keine *LVs*) manipuliert werden.[6] Es kann hierbei ebenfalls dazu kommen, dass die schreibenden Operationen nur `unvollständig` ausgeführt werden, z. B. indem Objekte nicht korrekt vernetzt werden oder Attributwerte nur inkonsistent geändert werden. Auch ist es möglich, dass eigentlich nicht zu modifizierende Objekte fälschlicherweise verändert werden.

Löschende Anteile fehlerhaft Sollen Elemente im Modell gelöscht werden, stellt dies eine dritte potentiell fehleranfällige Manipulation dar. Auch hierbei kann es potentiell vorkommen, dass eigentlich zu löschende Elemente nicht entfernt werden, oder dass nicht zu löschende Elemente gelöscht werden.

Expression Im `expressions`-Paket des *SDM*-Metamodell, s. Abbildung E.5 in Anhang E, sind die verschiedenen Unterklassen der `Expression`-Klasse definiert. Dabei weist jeder spezielle Untertyp Eigenschaften auf, die Ihn anfällig für bestimmte Fehlerarten machen.

`ObjectVariableExpression` Die `ObjectVariableExpression` referenziert eine *OV* und steht dafür, dass an dieser Stelle das Objekt, an welches die *OV* gebunden ist, eingesetzt werden soll. Offensichtlich liegt dann ein potentieller Fehler vor, wenn die referenzierte *OV* zu diesem Zeitpunkt ungebunden ist. Soll ausgedrückt werden, dass unter gewissen Umständen bei einem Stoppknoten `null` zurückgegeben wird, weil beispielsweise kein entsprechendes Modellelement gefunden werden kann, sollte immer ein entsprechendes Literal statt einer ungebundenen *OV* verwendet werden.

`LiteralExpression` Je nach der Situation der Verwendung, kann es schon einen Fehler darstellen, wenn des `null`-Literal verwendet wird. Dagegen ist eine `LiteralExpression` ohne Inhalt stets ein Fehler. Ansonsten wird der Wert einer `LiteralExpression` im Rahmen der Codegenerierung nicht weiter analysiert, was dazu führen kann, dass das Generat zu illegalem, nicht analysierbarem Code führt (statisch überprüfbar). Falls der Inhalt aus einem Java-Fragment besteht, können darüber hinaus typische Java-Fehler auftreten, die hier aber nicht weiter aufgeschlüsselt werden sollen.

`AttributeValueExpression` Eine `AttributeValueExpression` referenziert den Attributwert, der zu einem durch eine *OV* referenzierten Objekt aus dem Modell gehört. Typischer Fehler in diesem Zusammenhang sind, dass (i) die referenzierte *OV* zum Zeitpunkt der Auswertung ungebunden ist, (ii) die falsche *OV* referenziert wurde oder dass (iii) das ausgewählte Attribut vom falschen Typ ist, was im Falle

[6] Assoziationsinstanzen stellen hier keine „First-Class-Artifacts“ mit eigenständiger Identität und unabhängigem Lebenszyklus dar.

der Generierung von Java-Quellcode spätestens durch den Java-Compiler gemeldet werden würde.

MethodCallExpression Bei `MethodCallExpression`s kann es ebenfalls vorkommen, dass die Instanz, auf der die Operation bzw. Methode aufgerufen werden soll, undefiniert ist, da die entsprechende *OV* ungebunden ist. Ein anderer Fehler besteht darin, dass eine Operation mit inkompatiblem Rückgabetyp ausgewählt wurde. Auch das Vertauschen von Parametern bei der Definition der Wertebelegungen ist leicht möglich. Dies würde bei typkompatiblen Parametern auch nicht weiter auffallen, und stellt somit einen potentiell leicht zu übersehenden Fehler dar, der erst zur Laufzeit zu beobachten wäre.

Da zur Bestimmung eines Parameterwertes wiederum eine `Expression` verwendet werden kann, besteht eine weitere Fehlerklasse darin, dass diese eingebettete `Expression` selbst fehlerhaft ist.

ParameterExpression Instanzen dieser `Expression`-Art referenzieren einen der Parameter der zu implementierenden Operation bzw. stehen für dessen konkreten Wert zum Zeitpunkt der Auswertung respektive des Aufrufs. Problematisch ist hierbei der Fall einer Verwechslung. Diese manifestiert sich darin, dass ein anderer Parameter referenziert wird, als der eigentlich gewünschte.

Zusammenfassung In Abbildung 8.1 ist eine Übersicht des bis hierhin entwickelten Fehlermodells für die *SDM*-Sprache gegeben. Deutlich erkennbar sind die drei wesentlichen Teilaspekte – Knoten, Expresssions und der Kontrollfluss – die in ihrer Gesamtheit die *SDM*-Sprache ausmachen. Die wichtigen Fehlertypen des Graphtransformationsteils der Sprache, die unterhalb des mit `StoryNode` beschrifteten Knotens einen eigenen Teilbaum bilden, sind aus Platzgründen in dieser Darstellung ausgespart. Die Tabelle 8.1 ist dediziert den entsprechenden Details gewidmet. Zur kompakteren und übersichtlicheren Darstellung der Klassifikation, vgl. den linken Bereich der Tabelle, erfolgt die notwendige Fallunterscheidung zwischen *LVs* und *OVs* im rechten Teil der Tabelle anhand zweier separater Spalten.

	Link-Variablen	Object-Variablen
StoryNode		
Regeldefinition		
Variablendefinition		
Typ falsch	falscher Typ bei LV	falscher Typ bei OV
BindingOperator fehlerhaft	CHECK_ONLY vs CREATE vs DESTROY	CHECK_ONLY vs CREATE vs DESTROY
BindingSemantics fehlerhaft	MANDATORY vs NEGATIVE vs OPTIONAL	MANDATORY vs NEGATIVE vs OPTIONAL
Regelumfang		
benötigtes Element fehlt	LV vergessen	OV vergessen
nichtbenötigtes Element vorhanden	mind. eine LV ist überflüssig	mind. eine OV ist überflüssig
spezifische OV-Fehler		
BindingState falsch	---	BOUND vs UNBOUND
BindingExpression fehlerhaft	---	vgl. Fehler durch Expressions
inkompatible Typen bei BindingExpression	---	Typumwandlung bei inkompatiblen Typen
BindingExpression liefert null	---	Typumwandlung auf null angewendet
falsche Attributbedingung	---	falscher Operator, Vergleichswert etc.
Regelsemantik		
Verletzung der Injektivität durch gebundene OVs	---	×
inkonsistente Regelteile	×	×
Regel ergibt keinen validen Suchplan	×	×
Graphersetzung		
erzeugender Anteil		
unvollständig	• vertauschte Richtung • mind. ein Ende falsch • Linkerzeugung fehlt	• Objekterzeugung fehlt • Containment-Hierarchie nicht sichergestellt • potentiell unerreichbaren Knoten erzeugt • AttributeAssignment falsch • AttributeAssignment fehlt
falsche Elemente	Link vom falschen Typ	Objekt vom falschen Typ
modifizierender Anteil		
unvollständig	---	• Objekt(e) nicht richtig verbunden • Attribute unvollständig modifiziert
falsche Elemente	---	• falsches Objekt verschoben • falsches Attribut modifiziert
löschender Anteil		
unvollständig	• Links ohne Knoten gelöscht • Link(s) nicht gelöscht	Objekt(e) nicht gelöscht
falsche Elemente	falsche Kante entfernt	falsches Objekt gelöscht

Tabelle 8.1: Der Teil des *SDM*-Fehlermodells für `StoryNode`s („---“: „trifft auf Variablen dieser Art nicht zu“, „✗“: „Variablenart ist daran beteiligt“)

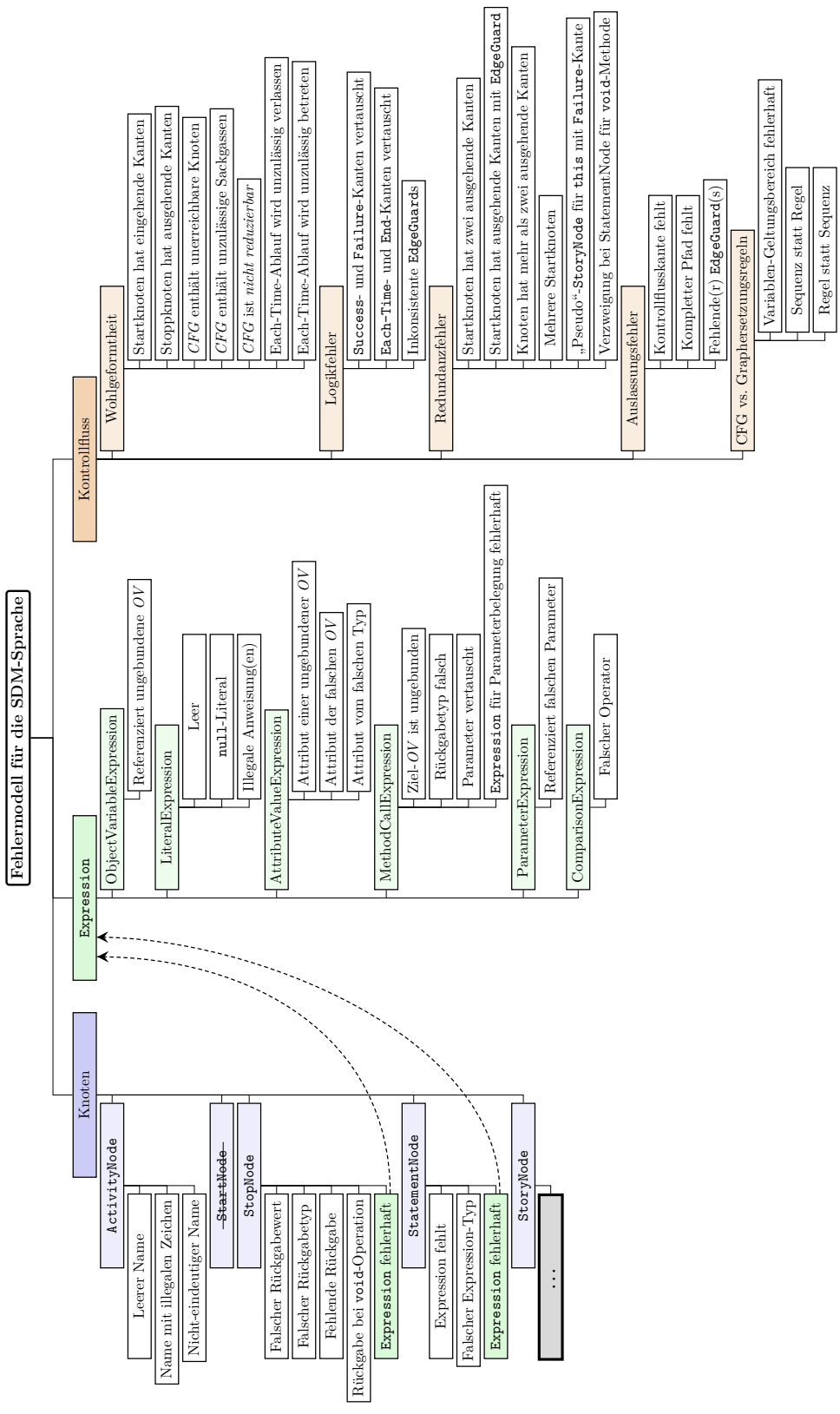


Abbildung 8.1: Fehlermodell zur Klassifikation *SDM*-typischer Fehler

8.3.2 Mutationsoperatoren für SDM-Transformationen

Nach der Analyse der Fehlerarten bei *SDM*-Transformationen, die in der Vorstellung des Fehlermodells aus Abbildung 8.1 mündeten, betrachten wir im Folgenden konkrete Mutatoren, vgl. Definition 5.24, für die Sprache. Diese sind in drei Hauptkategorien unterteilt, nämlich in Mutatoren zur Einbringung von synthetischen Fehlern hinsichtlich (i) der Struktur des Kontrollflusses, (ii) der Eigenschaften einzelner Aktivitätsknoten sowie (iii) der Graphtransformationsschritte.

Kontrollfluss

Im Folgenden werden die Mutationsoperatoren vorgestellt, die Fehler mit Bezug zum *SDM*-Kontrollfluss in das Programm einbringen können.

AddStopNodeForUnguardedTransition Der *AddStopNodeForUnguardedTransition*-Operator kann für jede `StoryNode`- oder `StatementNode`-Instanz angewendet werden, die nur eine ausgehenden Kontrollflusskante ohne Bedingung besitzt, was einer ausbleibenden Verzweigung des Kontrollflusses entspricht. Bei einer Anwendung wird die bestehende Kontrollflusskante mit der Bedingung `Success` oder `Failure` versehen. Darüber hinaus wird eine zweite Kontrollflusskante, mit der jeweils komplementären Bedingung zur zuvor gewählten, erzeugt, die den betrachteten Knoten mit einem ebenfalls neu erzeugten Stoppknoten verbindet. Je nach Rückgabetyp der Operation, muss der Stoppknoten auch einen Ausdruck zur Rückgabe eines Wertes spezifizieren. Für einfache Datentypen wird ein passender Standardwert (z. B. das ‚0'-Literal bei numerischen Typen oder der leerer String bei Zeichenketten) und für komplexe Typen `null` zurückgegeben.

Abbildung 8.2 verdeutlicht das Prinzip anhand eines Ausschnitts der Beispieltransformation. Für den Knoten `find_block_collector` werden hierbei ein Stoppknoten und eine neue Kontrollflusskante erzeugt sowie entsprechende Bedingungen an den Kanten eingeführt. Die Motivation für diesen Mutationsoperator ist, dass mit seiner Hilfe der ursprüngliche Verzicht auf eine Kontrollflussverzweigung sabotiert wird. Je nach Verteilung der Bedingungen an den Kanten wird der Ablauf gegebenenfalls vorzeitig abgebrochen und die Operation frühzeitig verlassen. Dadurch wird der Ablauf für bestimmte Konstellationen verkürzt, was durch das Testen aufgedeckt werden sollte.

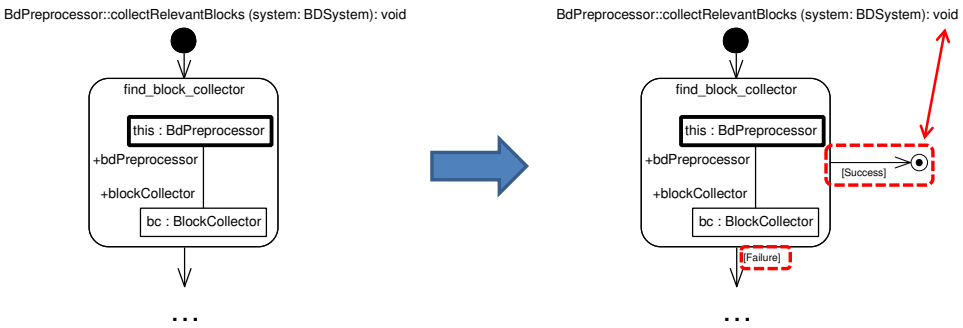


Abbildung 8.2: *AddStopNodeForUnguardedTransition* am Beispiel

SwitchSuccessAndFailureConditions Der Operator *SwitchSuccessAndFailureConditions* ist auf jede `StoryNode`-Instanz, die nicht die For-Each-Semantik besitzt, bzw. auf jede `StatementNode`-Instanz anwendbar, falls der entsprechende Knoten zwei ausgehende Kontrollflusskanten aufweist. Bei Anwendung sorgt der Mutator dafür, dass die `Success`- und `Failure`-Guards der beiden ausgehenden Kanten jeweils miteinander vertauscht werden. Eine günstige Eigenschaft ist, dass die Implementierung des Operators relativ trivial ist. Nachteilig wirkt sich eventuell die Tatsache aus, dass so eingebrachte Fehler ggf. durch umfangreiche statische Analysen, die den Gültigkeitsbereich von Variablen eingehend überprüfen, bereits vor der Testausführung entdeckt werden.[7] Schließlich sind zu bindende Variablen nach einem Fehlschlagen der Mustersuche als nicht gebunden anzusehen. Setzt man im weiteren Verlauf diese Variablen trotzdem als gebunden voraus, verletzt man offenkundig diese Annahme. Insgesamt ist dieser Mutator als eher einfach anzusehen. Dennoch simuliert er einen sehr häufig anzutreffenden Flüchtigkeitsfehler.

ConvertForEachToRegularPattern Im Rahmen einer von mir betreuten studentischen Arbeit [Pil14] wurden einige Mutatoren für die *SDM*-Sprache implementiert. Einer dieser Mutatoren bezieht sich beispielsweise auf einzelne For-Each-Knoten mit vorhandenem Each-Time-Zweig. Die Motivation für den *ConvertForEachToRegularPattern*-Operator besteht darin, zu simulieren, dass statt einem For-Each-Knoten versehentlich ein normaler Story-Knoten verwendet wird. Umgesetzt wird dies dadurch, dass der For-Each-Knoten zu einem normalen Story-Knoten konvertiert wird. Das alleine führt allerdings zu einer inkonsistenten Spezifikation, da die ausgehenden Kanten noch die falschen Wächterausdrücke tragen. Folglich müssen noch die `Each-Time`-Kante zu einer `Success`-Kante und die `End`-Kante zu einer `Failure`-Kante umgewandelt werden. Zusätzlich werden die rücklaufenden Kanten aus dem vormaligen Each-Time-Zweig, die in den betrachteten Knoten einlaufen, auf den Zielknoten der ehemaligen `End`-Kante „umgebogen“. Anhand des Beispiels aus Abbildung 8.3 lassen sich diese Schritte an einem Ausschnitt der Operation `createSystemToMethodMappings` (vgl. Anhang A.3.3) leicht nachvollziehen.

Aktivitätsknoten

Mutatoren, die sich auf solche Aktivitätsknoten beziehen, die keine Story-Knoten sind, sind hier eher von untergeordnetem Interesse, da sich diesbezügliche Fehlerquellen in erster Linie auf `Expression`s beziehen. Dennoch werden hier drei relativ leicht umsetzbare Mutatoren – die im Rahmen dieser Arbeit auch implementiert wurden – vorgestellt.

ReturnDefaultValueOnStopNode Der *ReturnDefaultValueOnStopNode* genannte Mutator modifiziert Stoppknoten, für welche ein Rückgabewert spezifiziert ist, dergestalt, dass der Knoten statt dem ursprünglichen Rückgabewert nur noch ein fixes Literal, das einem Standardwert für den jeweiligen Rückgabetypen der Operation entspricht, zurückliefert. (Die ursprüngliche Version des Mutators, die ebenfalls im Rahmen von [Pil14] entstand, ersetzte den Rückgabewert nur bei Operationen mit benutzerdefiniertem Typ als Rückgabetyp durch das `null`-Literal.) Der Mutator simuliert folglich Fehler, die sich

[7] Unter Umständen kann es für die Bewertung von Testmengen durch die Mutationsanalysemethodik also sinnvoll sein, bestimmte statische Analysen zu deaktivieren, um so überprüfen zu können, wie sich die Testmenge für entsprechende Fehlerfälle schlagen würde.

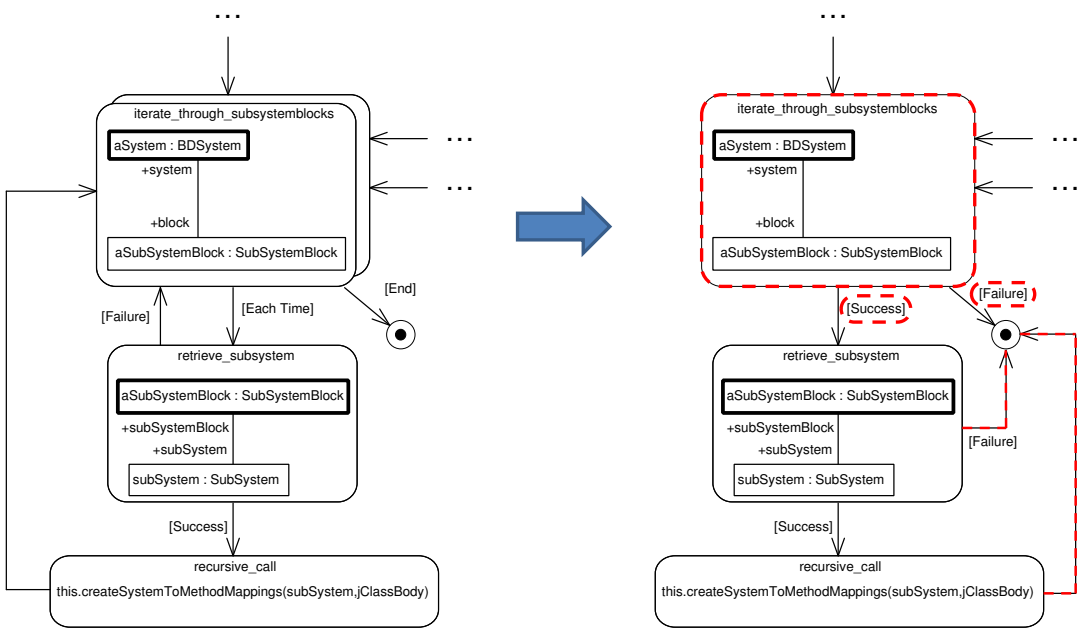


Abbildung 8.3: *ConvertForEachToRegularPattern* am Beispiel

darin äußern, dass ein falscher Wert respektive nur der Standardwert als Ausgabe der Operation (an einem bestimmten Ausgabepunkt) zurückgeliefert wird. In Abbildung 8.4 ist ein Beispiel auf Basis eines Ausschnitts der `getTopmostAssignment`-Operation aus Anhang A.3.3 gegeben. Wichtig ist hierbei, dass die Mutation nicht bei Operationen mit dem Rückgabetyp `void` ausgeführt wird. Außerdem sollte an den entsprechenden Anwendungsstellen sichergestellt sein, dass nicht ein Mutant entsteht, der von vornherein äquivalent ist, weil z. B. der Stoppknoten bereits `null` oder den jeweilig passenden Standardwert zurückliefert.

Abbildung 8.4: *ReturnDefaultValueOnStopNode* am Beispiel der `getTopmostAssignment`-Operation

ChangeStatementNodeParameterBinding Als ein Beispiel dafür, wie ein Operationsaufruf innerhalb eines `StatementNodes` mutiert werden kann, betrachten wir hier den *ChangeStatementNodeParameterBinding*-Operator, der in einer ersten Variante ebenfalls im Rahmen der Arbeit [Pil14] umgesetzt wurde. Dieser Mutationsoperator wird auf

`StatementNode`-Instanzen angewendet, falls diese den Aufruf einer Operation *mit Parametern* umfassen. Dabei wird die Variablenreferenz der Wertebelegung eines Parameters durch eine andere ersetzt. Im hier betrachteten Fall muss sich folglich der Wert mindestens eines der beteiligten Parameter durch eine `ObjectVariableExpression` ergeben, da nur Wertebelegungen durch *OVs* berücksichtigt werden. Der *Wert* des Parameters bestimmt sich somit zum Modellelement, an das die referenzierte *OV* zum Zeitpunkt des Aufrufs gebunden ist. Zwar können grundsätzlich auch andere `Expression`-Untertypen (`AttributeValueExpression`, `ParameterExpression` und `ComparisonExpression`) dazu benutzt werden, um den Wert eines Parameters festzulegen. Diese Fälle sollen aber ausgespart werden, da die Festlegung von Parameterwerten mittels *OVs* den typischen Fall bei *SDM*-Transformationen darstellt.

Wird eine potentielle Anwendungsstelle für den Mutator entsprechend der oben aufgeführten Kriterien entdeckt, muss zusätzlich noch geprüft werden, ob eine andere *OV* vom passenden Typ im Diagramm vorkommt. Die geeigneten Typen ergeben sich unmittelbar aus dem Typ des Operationsparameters (als kompatibel anzusehen sind der Typ der Parameters selbst sowie sämtliche Unterklassen). Um die Komplexität der Generierung entsprechender Mutanten überschaubar zu halten und um potentiell mögliche Fehler nicht auszuklammern, wird nicht weiter überprüft, ob die geeignet erscheinenden *OV*-Kandidaten zum Zeitpunkt des Operationsaufrufs auch tatsächlich gebunden sind bzw. überhaupt sein können. Auch wäre es grundsätzlich denkbar, beispielsweise auszuschließen, dass eine bestimmte *OV* als Substitut genutzt wird, falls sie schon den Wert eines anderen Parameters definiert – folglich eine *OV* also mehrfach in einem Aufruf vorkommen würde, was einem Entwickler vielleicht eher als fehlerhaft auffallen könnte. Andererseits würden verschiedene *OVs* auch nicht garantieren, dass die tatsächlich übergebenen Parameter*werte* zwingend verschieden sind. Grundsätzlich besteht also die Gefahr, dass durch diesen Operator semantisch äquivalente Mutanten generiert werden.

In Abbildung 8.5 ist die Anwendung des Mutationsoperators auf den `StatementNode` „recursive_call“ gezeigt. Der Parameter vom Typ `BDSystem` wird in der Ausgangssituation durch den Wert der *OV* `subSys` definiert. Letzterer ergibt sich aus der Auswertung des ebenfalls gezeigten For-Each-Knotens. Nach der Anwendung des Operators bestimmt sich der Wert des Parameters zum Wert der *OV* `system`, welche ebenfalls im Diagramm vorkommt. Für das konkrete Beispiel würde es hierdurch zu einer endlosen Rekursion kommen.

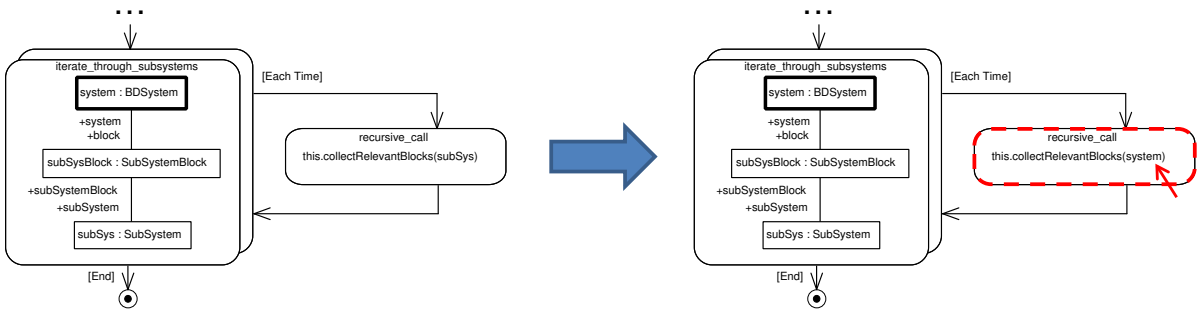


Abbildung 8.5: *ChangeStatementNodeParameterBinding* am Beispiel der `collectRelevantBlocks`-Operation

ReplaceParameterBindingByDummyLiteral Neben der Gefahr, sich bei der Formulierung einer `MethodCallExpression` bezügliche der Auswahl der richtigen *OV* zu irren, kann einem Entwickler auch leicht der Fehler unterlaufen, dass er es vergisst oder übersieht, einen Parameter richtig zu initialisieren. Beispielsweise ist es in dem verwendeten *SDM*-Editor relativ leicht möglich, dass man gar keinen Wert für einen Parameter festlegt und folglich entweder ein Standardwert der Zielsprache genutzt wird oder kein funktionsfähiger Code entsteht. Entsprechende Fehler lassen sich mit dem *ReplaceParameterBindingByDummyLiteral*-Mutator simulieren. Dazu werden für jede `StatementNode`-Instanz die Parametervorgaben untersucht und für alle Parameter, die *nicht* mittels `LiteralExpression` belegt werden, ein entsprechender Mutant erzeugt. Dabei wird der ursprüngliche Ausdruck, der den Wert des Parameters bestimmt, durch eine `LiteralExpression`-Instanz ersetzt, die als Literal einen Standardwert für den entsprechenden Parametertyp umfasst. So wird für Nicht-Standardtypen das `null`-Literal verwendet, für String-Parameter ein Literal, das den leeren String repräsentiert und für andere primitive Typen entsprechende Initialwert-Literale. Der Mutationsoperator lässt sich relativ leicht umsetzen. Zu beachten ist hier nur, dass Literalausdrücke selbst nicht ersetzt werden sollen, da man ansonsten diese Ausdrücke recht aufwendig analysieren bzw. interpretieren müsste, um äquivalente Mutanten mit ausreichend hoher Wahrscheinlichkeit, näherungsweise ausschließen zu können.

Für das Beispiel aus Abbildung 8.5 ergibt sich bezüglich der *ReplaceParameterBinding-ByDummyLiteral*-Mutation die selbe Anwendungsstelle. Das Ergebnis unterscheidet sich allerdings: der Wert des Parameters zum Aufruf der `collectRelevantBlocks`-Operation ergibt sich in diesem Fall nämlich zu `null`.

Graphersetzung

Für die Bewertung der Leistungsfähigkeit des *RPC*-Ansatzes sind die bisher vorgestellten Mutationsoperatoren für den Kontrollfluss sowie die Nicht-StoryNode-Knoten nicht ausreichend. Der Fokus des Testens liegt ja gerade auf den spezifischen Fehlern des anfälligen Graphtransformationsanteils der *MT*. Somit werden Mutationsoperatoren benötigt, die speziell Fehler in den Inhalt der `StoryNode`-Instanzen einbringen können. Einige solcher Operatoren werden im Folgenden beschrieben, grob unterteilt in Mutatoren für die beiden strukturellen Elemente, aus denen Regeln aufgebaut sind: *OVs* und *LVs*.

Object-Variablen Bezogen auf *OVs* gibt es verschiedene Aspekte, die prinzipiell durch Mutationsoperatoren manipuliert werden können, wie (a) der Typ, (b) die Binding-Semantik, (c) der Binding-Zustand, (d) der Binding-Operator, (e) Attributbedingungen, (f) Attributzuweisungen sowie (g) Binding-Ausdrücke zum Binden der Variable an einen bestimmten Wert. Mit Ausnahme des letztgenannten Punktes werden für die anderen Aspekt im Folgenden konkrete Mutationsoperatoren beschrieben.

SubstituteTypeBySubtype Der *SubstituteTypeBySubtype*-Operator überprüft für jede ungebundene *OV*, ob für den jeweiligen Typ der Variable im Metamodell Unterklassen existieren, die als Substituts des ursprünglichen Typs dienen können. Falls dies gegeben ist, wird für jede Variable und jede verfügbare Unterklasse eine mutierte Variante der Transformation erzeugt, indem jeweils der Typ einer einzelnen *OV* auf eine der passenden Unterklassen eingeschränkt wird. Da Unterklassen ihren Basistypen grundsätzlich

nur um zusätzliche Eigenschaften ergänzen, ist jede Unterklasse ein geeigneter Substitutionskandidat.

Problematisch kann bei einer naiven Umsetzung des Mutationsoperators allerdings sein, dass ohne weitere Maßnahmen die veränderte *OV* an anderen Stellen der Transformation noch mit der ursprünglichen Typdefinition vorkommen kann, z. B. auch in gebundener Form. Um das Ergebnis der Typersetzung robuster gegenüber Inkonsistenzen und Widersprüche bezüglich der Variablendefinitionen zu machen, wurde auch ein verfeinerter Ansatz verfolgt und umgesetzt. Bei diesem werden zusätzlich *alle weiteren Verwendungsstellen* der betrachteten Variable innerhalb der Operation bestimmt und jede Verwendungsstelle entsprechend angepasst. Andererseits ist das Problem solcher Inkonsistenzen in der Praxis, bezogen auf den aktuellen Stand der Implementierung des Transformationswerkszeugs eMoflon, nicht so gravierend, wie es auf den ersten Blick eventuell erscheinen mag. Der Grund hierfür liegt darin, dass die *erstmalige* (in den relevanten Fällen impliziert dies auch die *ungebunden*) Verwendung einer *OV* den Typ der *OV* für die restliche Transformation determiniert. Nachfolgende Typredefinitionen für bereits verwendete *OVs* werden de facto ignoriert. Andererseits ignoriert der hier genutzte *SDM*-Editor diese Angaben nicht, so dass die nutzbaren Linktypen stets in Abhängigkeit des spezifizierten *OV*-Typs zur Auswahl angeboten werden. Dadurch ist es möglich, dass dann „falsche“ Kanten hinsichtlich des Typs spezifiziert werden, falls sich die spezifizierten Typen hierin unterscheiden. Fehler durch inkompatible Links sind allerdings im Falle des *SubstituteTypeBySubtype*-Operators aufgrund des Subtyppolymorphismus sowieso ausgeschlossen. Und auch unter anderen Umständen würden diese spätestens bei der Kompilierung des Generats mit sehr hoher Wahrscheinlichkeit auffallen.

SubstituteTypeBySupertype Das Gegenstück des zuvor vorgestellten Mutationsoperators bildet der *SubstituteTypeBySupertype*-Mutator. Hierbei wird, wie der Name bereits andeutet, der Typ einer *OV* in Gestalt einer bestimmten Klasse durch entsprechend *kompatible* Oberklassen ersetzt. Die Kompatibilität der Oberklasse bestimmt sich dabei durch den Verwendungskontext der *OV*. Insbesondere angrenzende Links sowie Attributbedingungen und -zuweisungen schränken die Menge der tatsächlich nutzbaren Oberklassen ein, da entsprechende Assoziationen oder Attribute in einigen oder allen Oberklassen nicht definiert sein können.

Dabei besteht bezüglich der Typersetzung einer einzelnen *OV* grundsätzlich die gleiche Problematik bezüglich der Inkonsistenzen von Typangaben wie im vorherigen Fall. Allerdings ist es hier nicht möglich, einfach den Typ an allen Stellen, an denen die gerade betrachtete *OV* auftritt, zu ändern. Es könnte ja immer sein, dass an einer dieser Stellen eine weitere Bedingung die Anpassung mit einer Oberklasse gerade verhindert. Folglich müsste man für jede Stelle, an denen der Typ angepasst werden soll, individuell überprüfen, ob die Voraussetzungen dafür erfüllt sind; und die gesamte Menge an Modifikationen nicht ausführen, falls an einer Stelle die Bedingungen nicht erfüllt sein sollten. Um den Mutator nicht zu komplex werden zu lassen, wurde auf einer Umsetzung dieser Art verzichtet. Bei der hier verfolgten Herangehensweise wird ohne weitere Analysen probiert, den *OV*-Typ an einer Stelle isoliert auf eine Oberklasse zu verallgemeinern. Sollte dies an anderen Stellen in der Transformation zu Problemen führen, so wird das Generat insgesamt im besten Fall nicht kompilierbar sein und der Mutant wird von Beginn an als nicht überlebensfähig erkannt.

AlterOvBindingSemantics Eine weitere Gruppe von Mutatoren sorgt dafür, dass die Binding-Semantik einzelner *OVs* manipuliert wird. Als Anwendungsstellen des Mutators werden alle vorhandenen, ungebundenen Auftreten solcher Variablen bestimmt. Darüber hinaus werden keine weiteren Analysen dahingehend durchgeführt, ob das Ergebnis der hier rein syntaktischen Manipulation semantisch noch sinnvoll ist. Die folgenden Veränderungen – die auch in Abbildung 8.6 symbolisch für eine *OV* skizziert werden – sind möglich:

NegativeToOptional Eine als *NAC* konfigurierte *OV* wird zu einem optionalen Knoten umkonfiguriert, vgl. Abbildung 8.6, rechtwinkliger Pfeil unten links. Dadurch wird die *NAC* effektiv entfernt, ein vormals auszuschließendes Element aber nicht zwingend vorausgesetzt. Darüber hinaus sind keine weiteren Probleme zu erwarten, da die entsprechende *OV* sowieso nicht anderweitig verwendet werden sollte.

NegativeToMandatory Das *NAC*-Element wird zu einem obligatorischen Element des Matches, vgl. Abbildung 8.6, rechtwinkliger Pfeil unten rechts. Somit wird die Bedeutung der Variablen ins Gegenteil umgekehrt. Da ein valider Suchplan auch schon für die *NAC*-behaftete Variante existieren musste, ist auch nach der Umwandlung ein valider Suchplan gegeben, da ja das dann Nicht-Mehr-*NAC*-Element problemlos bei der Suche traversiert werden kann.

OptionalToNegative Umgekehrt können auch optionale *OVs* in *NACs* um gewandelt werden, vgl. Abbildung 8.6, innerer rechtwinkliger Pfeil unten links. Hierbei können allerdings unter Umständen fehlerhafte Spezifikationen entstehen, wenn im weiteren Verlauf auf das zuvor optionale und nun anschließend auszuschließende Element durch weiteres Auftreten der Variable zugegriffen wird.

OptionalToMandatory Die Umwandlung eines optionalen in einen obligatorischen Knoten, vgl. Abbildung 8.6, rechtsgerichtete, horizontale Pfeile, stellt keine große Herausforderung dar. Das Ergebnis sollte ein restriktiveres, aber grundsätzlich lauffähiges Programm sein.

MandatoryToNegative In diesem Fall wird eine normale *OV* zu einer *NAC* umkonfiguriert, vgl. Abbildung 8.6, innerer rechtwinkliger Pfeil unten rechts. Diese Umwandlung kann sehr weitreichende Konsequenzen nach sich ziehen. Insbesondere könnten durch die Mutation Muster bzw. Regeln entstehen, für die kein valider Suchplan mehr existiert. Außerdem kann nach einer solchen Mutation in nachfolgenden Regeln nicht mehr auf die ursprünglich zu bindende Variable zugegriffen werden. Um nicht überlebensfähige Mutanten sicher auszufiltern, müsste die Transformationsbeschreibung dahingehend intensiv analysiert werden. Andererseits kann einem Entwickler ein solcher Fehler relativ leicht unterlaufen, ohne dass dies direkt auffallen muss. Somit erscheint dieser Mutationsoperator auch in seiner einfachsten Umsetzungsvariante als sinnvoll.

MandatoryToOptional Bei der Umwandlung einer obligatorisch zu suchenden in eine optionale *OV*, vgl. Abbildung 8.6, linksgerichtete, horizontale Pfeile, kommt es stark auf den konkreten Einzelfall an, um entscheiden zu können, ob das Mutationsergebnis noch semantisch sinnvoll ist. Die Wahrscheinlichkeit für einen überlebensfähigen Mutanten ist allerdings deutlich größer als im vorherigen Fall.

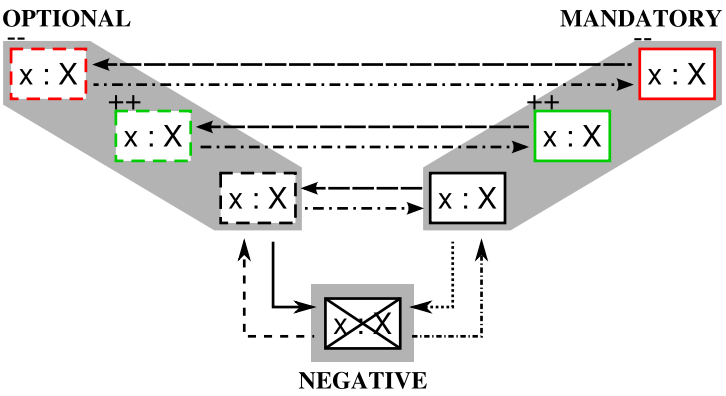


Abbildung 8.6: Mögliche Änderungen durch `BindingSemantics`-Mutationen (hier nur für die Fälle mit ungebundenen *OVs*)

AlterOvBindingState Auch der Bindungszustand, der durch den Entwickler direkt spezifiziert wird, hat großen Einfluss auf die Semantik und die Auswertung einer Regel. Folglich sollten Tests existieren, die sicherstellen, dass dieser wichtige Teil der Spezifikation korrekt ist. Dazu lassen sich durch entsprechende Mutationsoperatoren leicht Fehler in die Spezifikation einbringen, indem die `BindingSemantics`-Eigenschaft einer *OV* modifiziert wird. Da es nur zwei Zustände gibt, s. Abbildung E.6, sind genau zwei mögliche Wechsel denkbar, für die hier jeweils ein eigener Mutationsoperator zuständig ist.

BoundToUnbound Standardmäßig wird eine *OV* vom Editor in ihrer ungebundenen Variante angelegt. Soll sie als gebunden modelliert werden, muss der Entwickler dies explizit spezifizieren. Dies kann natürlich auch vergessen werden, was einen relativ häufig gemachten Fehler darstellt. Leider ist der Fehlerzustand auch nicht immer unmittelbar erkennbar, da das Ergebnis weiterhin funktionsfähig ist und sich, je nach Modell, sogar äquivalent verhalten kann. Ein passender Mutationsoperator, der auch umgesetzt wurde, identifiziert gebundene *OVs*, deren Namen ungleich „this“ ist und die in einer Regel mit mindestens einer weiteren gebundenen *OV* vorkommen. Für jede solche Situation wird dann ein Mutant erzeugt, der sich nur im Bindungszustand dieser einen Variablen vom Originalprogramm unterscheidet. Gebundene Variablen, deren Wert sich durch einen Operationsparameter ergeben würde, werden dabei genauso behandelt, wie andere gebundene Variablen.

UnboundToBound Die entgegengesetzte Richtung, von ungebunden zu gebunden, ist ebenfalls grundsätzlich möglich. Allerdings ist ein Irrtum in diese Richtung aufgrund der Menüführung im Editor eher unwahrscheinlich. Zusätzlich würde ein solcher Fehler bei der Ausführung, je nach Verwendung der Variablen in anderen Regeln, eventuell durch Fehler im Generat auffallen. Allerdings ist dies in der aktuellen Codegenerierung keinesfalls so sicher, wie man eventuell vermuten würde. Aufgrund der geringen Auftrittswahrscheinlichkeit im Zusammenspiel mit der Chance auf fehlerhaften Code wurde auf eine Umsetzung im Rahmen der Arbeit verzichtet.

AlterOvBindingOperator Als dritte wesentliche Eigenschaft einer *OV* bildet auch der Bindungsoperator eine weitere potentielle Fehlerquelle. Durch die drei Optionen ergeben

sich insgesamt sechs Möglichkeiten, wie die Werte verändert werden können. In Abbildung 8.7 sind diese Übergangsmöglichkeiten für eine *OV* skizziert. Allerdings sind nicht alle Möglichkeiten gleich relevant, was die Gefahr unentdeckter Fehlerzustände angeht. Hier wurden nicht alle denkbaren Übergänge auch durch Mutatoren implementiert.

CheckToCreate Einen eher unwahrscheinlichen Fehlerfall stellt beispielsweise die Situation dar, dass bei einer *OV*, die eigentlich einen `CHECK_ONLY`-Bindungsoperator besitzen soll, der `CREATE`-Operator gewählt wurde. Vgl. dazu in Abbildung 8.7 die rechtwinkligen, nach unten zeigenden Pfeile im rechten Teil. Dennoch wurde dieser Fall als Mutationsoperator realisiert. Er kann nur auf ungebundene *OVs* ohne Attributbedingungen angewendet werden und sorgt dafür, dass im Rahmen der Regelauswertung ein entsprechendes Objekt im Modellgraphen neu angelegt wird, anstatt dass es gesucht wird.

CheckToDestroy Ändert man den Bindungsoperator für eine *OV* von `CHECK_ONLY` dagegen auf `DESTROY` ab, vgl. Abbildung 8.7, rechtwinklige, nach unten zeigende Pfeile im linken Teil, so wird das entsprechende Modellelement zwar immer noch gesucht. Im Falle, dass eine Entsprechung gefunden wird, wird dieses Element aber aufgrund der Mutation nun gelöscht. Auch dieser Fall wurde im Rahmen der Arbeit durch einen entsprechenden Mutationsoperator abgedeckt. Wobei diesbezüglich auch die zuvor getroffene Feststellung Bestand hat, dass entsprechende Fehler in der Praxis eher seltener anzutreffen sind, zumindest im Vergleich zu den beiden zwei ebenfalls umgesetzte und noch ausstehenden Mutationsoperatoren, die sich auch auf den Bindungsoperator beziehen.

CreateToCheck Soll ein Objekt im Modell erzeugt werden, so muss die entsprechende *OV* mit dem `CREATE`-Operator versehen sein. Dies kann allerdings leicht vergessen werden, so dass anstatt des Anlegens eines neuen Objektes nach einem existierenden gesucht wird. Ein entsprechender Mutationsoperator, der auch implementiert wurde, simuliert diesen Fehler, indem für einzelne *OVs* mit Bindungsoperator `CREATE` auf `CHECK_ONLY` abgeändert wird, vgl. Abbildung 8.7, rechtwinklige, nach links zeigende Pfeile im rechten Teil.

CreateToDestroy Soll eine *OV* einen anderen als den standardmäßigen Bindungsoperator tragen, so muss entweder der `CREATE`-Operator, wie im Folgenden betrachtet, oder der `DESTROY`-Operator ausgewählt werden. Da beide Operationen auf die gleiche Art und Weise konfiguriert werden, ist es leicht möglich, dass man sich bei der Auswahl der Optionen irrt und die falsche Option selektiert. Folglich erscheint die Umsetzung eines entsprechenden Mutators, der diesen Fehler simuliert sinnvoll. Vgl. dazu auch in Abbildung 8.7 die beiden nach links gerichteten, horizontalen Pfeile. Dabei sind anzulegende *OVs* mit Attributzuweisungen ggf. auszuschließen, da letztere für einen dann zu löschenden Knoten als wenig sinnvoll erscheinen.

DestroyToCreate Der entgegengesetzte Fall zum vorherigen ist ebenso möglich. Somit ist ein Mutationsoperator, der den `BindingOperator` von `DESTROY` auf `CREATE` ändert, ebenfalls naheliegend, vgl. in Abbildung 8.7 die beiden nach rechts gerichteten, horizontalen Pfeile. Eine Besonderheit besteht allerdings für den Fall, dass die *OV* eigene Attributbedingungen aufweist. Dies macht für anzulegende Knoten semantisch keinen Sinn, und so erscheint es äußerst unwahrscheinlich, dass ein Benutzer

für einen Knoten mit Attributbedingungen tatsächlich den `CREATE`-Operator *unentdeckt* selektiert bzw. beibehält. Folglich ignoriert die hier umgesetzte Implementierung des Operators *OV*-Instanzen mit Attributbedingungen.

DestroyToCheck Der letzte noch ausstehende Fall betrifft *OVs*, die als zu löschend modelliert wurden. Grundsätzlich ist es möglich, dass vergessen wurde, entsprechende *OVs* auch so zu konfigurieren, dass der Knoten im Modell, so er denn gefunden wird, auch gelöscht wird. Diesen relativ naheliegenden Flüchtigkeitsfehler simuliert der *DestroyToCheck*-Operator durch ein entsprechendes Vertauschen des Bindungsoperators, vgl. in Abbildung 8.7 die rechtwinkligen, nach rechts zeigenden Pfeile im linken Bereich.

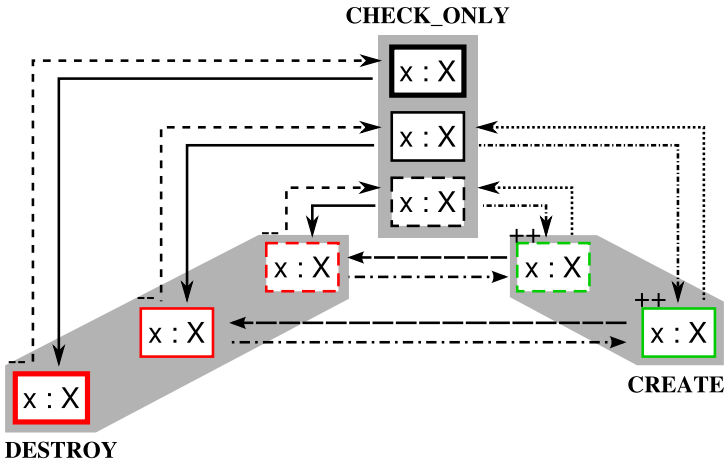


Abbildung 8.7: Mögliche Änderungen durch `BindingOperator`-Mutationen

AttributeCondition Attributbedingungen von *OVs* sind, wie bereits dargelegt, ebenfalls Quelle möglicher Fehler, sei es, weil sie fehlerhaft spezifiziert wurden oder weil sie inkorrekterweise komplett ausgelassen wurden. Somit sind verschiedene Mutationen denkbar, wobei hier die Darstellung auf drei konkrete Mutatoren beschränkt sein soll.

RemoveAllAttributeConditions Es ist unmittelbar offensichtlich, dass das Fehlen von eigentlich benötigten Attributbedingungen einen Fehlerfall darstellt. Genau dies simuliert der *RemoveAllAttributeConditions*-Operator, indem er *alle* Attributbedingungen einer einzelnen *OV* entfernt.

RemoveOneAttributeCondition Neben der Möglichkeit, dass Attributbedingungen komplett fehlen, kann es natürlich auch vorkommen, dass von mehreren benötigten Bedingungen nur einige wenige vergessen wurden. Der *RemoveOneAttributeCondition*-Operator simuliert diese Art von Fehler, indem er einzelne Attributbedingung (von mehreren vorhandenen) entfernt.

AlterComparisonOperatorInAttributeCondition Bei Attributbedingungen, die einen Vergleich mit einem anderen Wert mittels `ComparisonExpression` umfassen, kann es vorkommen, dass der falsche Vergleichsoperator ausgewählt wurde, vgl. hierzu den

`ComparisonOperator` in Abbildung E.5. Ein entsprechender Mutator bring Fehler dieser Art in die Beschreibung ein, indem entsprechende Operatoren innerhalb von Ausdrücken ausgetauscht werden. Dies entspricht dem Vorgehen eines der klassischen Mutationsoperatoren für imperative Programmiersprachen, wie beispielsweise unter der Bezeichnung „*Relational Operator Replacement*“ in [AO08, S. 183] vorgestellt.

AttributeAssignment Neben Bedingungen können *OVs* auch Zuweisungen für Attribute umfassen. Die folgenden Mutatoren stellen diesbezüglich naheliegende Optionen dar, von denen hier allerdings nur ein Operator stellvertretend umgesetzt wurden.

RemoveAllAssignments Wie Attributbedingungen müssen auch Zuweisungen explizit angegeben werden. Dies kann also ebenfalls unbeabsichtigt vergessen werden, was dazu führen kann, dass betroffenen Attribute nicht angepasst bzw. initialisiert werden. Der *RemoveAllAssignments* genannte Operator entfernt zur Erzeugung entsprechender Fehler alle vorhandenen Attributzuweisungen einer einzelnen *OV* einer Regel. Der Operator wurde im Rahmen der Arbeit umgesetzt und im Folgenden verwendet.

RemoveOneAssigment Anstatt alle Zuweisungen zu entfernen, können selbstverständlich auch nur einzelne Zuweisungen entfernt werden. Dies kann beispielsweise durch den hier nicht implementierten *RemoveOneAssigment*-Operator erfolgen.

AssignDefaultValue Neben dem Entfernen von Zuweisungen können bestehende Zuweisungen auch manipuliert werden, um dadurch falsche Zuweisungsausdrücke zu simulieren. Der hier ebenfalls ausgesparte *AssignDefaultValue*-Operator ändert dazu einzelne Zuweisung so ab, dass der ursprüngliche Ausdruck, durch eine Wertzuweisung eines festen Standardwerts des entsprechenden Typs respektive des `null`-Literals, ersetzt wird.

Sonstige Im Folgenden sind noch einige weitere Mutationsoperatoren skizziert, die sich nicht den zuvor beschriebenen Gruppen zuteilen lassen. Eine Implementierung im Rahmen der Arbeit war aus Zeitgründen nicht mehr möglich.

BindingExpression-Mutationen Einen Sonderfall bezüglich der Verwendung einer *OV* stellt das explizite Binden an ein per Ausdruck referenziertes Objekt dar, vgl. die `bindingExpression`-Referenz zwischen `ObjectVariable` und `Expression` in Abbildung E.6. Der konkret verwendete Ausdruck kann je nach Typ und Verwendungszweck (Typecast, Methodenaufruf, Typüberprüfung etc.) auf unterschiedlichste Arten mutiert werden. Beispielsweise könnte die Quelle der Zuweisung ausgetauscht werden. Oder ein Methodenaufruf erfolgt auf einer andere *OV* als Ziel des Aufrufs.

RemoveOv Vergleichbar zum Entfernen ganzer Anweisungen in klassischen Programmen, kann bei *SDM*-Transformationen auch eine einzelne *OV* aus einer Regel entfernt werden. Je nach Vernetzungsgrad der Variablen müssten zusätzlich noch angrenzende Kanten entfernt werden. Gegebenenfalls macht eine weitere Differenzierung entsprechender Operatoren anhand der Eigenschaften der zu löschenden *OV* Sinn.

So ist das Entfernen gebundener Knoten tendenziell problematischer für die Überlebenschancen der resultierenden Mutanten, als das Entfernen ungebundener Knoten, da im ersteren Fall bereits statisch kein valider Suchplan mehr zu bestimmen sein kann. Ungebundener Knoten lassen sich dagegen schwerer entfernen als optionale.

AddUnconnectedOv Das Gegenstück zum Löschen einer *OV* stellt das Neuanlegen und Hinzufügen von weiteren *OVs* dar. So können beispielsweise in einer Regel isolierte Knoten neu angelegt werden. Für einen solchen *AddUnconnectedOv*-Operator sind verschiedene Varianten des in Abhängigkeit der *OV*-Eigenschaften denkbar.

AddConnectedOv Der *AddConnectedOv*-Operator erzeugt dagegen nicht nur eine neue *OV*, sondern verbindet diese auch gleich noch per neuer *LV* mit anderen Elementen der ursprünglichen Regel. Hierbei ergeben sich durch die möglichen Eigenschaften der dazu notwendigen *LV* weitere Variationsmöglichkeiten. Was hinsichtlich solcher Fehler als realistisch anzusehen ist, muss hier allerdings offen bleiben.

Link-Variablen Bis jetzt wurden *OV*-spezifische Mutationsoperatoren vorgestellt. Nun kommen wir zu Mutatoren mit Bezug zu *LVs*. Aufgrund des geringeren Umfangs an charakteristischen Eigenschaften sind diesbezügliche Fehlerarten nicht so vielseitig, was sich hier auch in einer geringeren Anzahl an Mutationsoperatoren zeigt.

SubstituteLvType Beim Anlegen einer *LV* werden durch diese maximal zwei distinkte *OVs* verbunden. Durch die Typen des Quell- und des Zielknotens wird die Auswahl möglicher *LVs* eingeschränkt. Trotzdem kommt es in der Regel vor, dass mehrere *LV*-Typen gleichzeitig möglich sind, was dem Benutzer die Freiheit lässt, sich auch für eine falsche Option zu entscheiden. Um entsprechende Fehler zu simulieren, erzeugt der *SubstituteLvType*-Operator für jede *LV* und jeden kompatiblen Link-Typen einen Mutanten, bei dem der ursprüngliche Kantentyp durch den kompatiblen Typ ersetzt wird. Dies umfasst als Spezialfall auch die Funktionalität des nachfolgend beschriebenen *ReverseLvDirection*-Mutators.

ReverseLvDirection Beim Anlegen einer *LV* kann die Situation auftreten, dass beide *OVs* jeweils die Quell- *und* die Zielrolle einnehmen können. In diesen Fällen ist die Beziehung in beide Richtungen möglich, wobei sich der Entwickler für eine Richtung entscheiden muss. Hierbei besteht die Gefahr, sich falsch zu entscheiden, ohne dass dies unmittelbar auffallen muss. Der *ReverseLvDirection* genannte Operator simuliert Fehler dieser Art, indem er passende Situationen sucht, in denen vorhandene *LVs* auch in entgegengesetzter Richtung valide wären und einen Mutanten durch ein Umkehren der Richtung erzeugt. Bei der hier genutzten Implementierung wird die Funktionalität des Operators vollständig vom *SubstituteLvType*-Operator mit abgedeckt, da auch *LVs* vom gleichen Typ, aber in umgekehrter Richtung, als Substitut genutzt werden.

RemoveLv Wie auch schon bei *OVs* können Fehler sehr leicht dadurch entstehen, dass eigentlich benötigte Entitäten in der Regel vergessen werden. Durch das Entfernen einzelner *LVs* kann dies simuliert werden. Das Hauptproblem bei dieser Art der Mutation besteht darin, dass bei einer naiven Implementierung eine große Anzahl an Mutanten

generiert werden würden, die zum Teil nicht überlebensfähig hinsichtlich der Suchplangenerierung wären. Ein deutlicher Anteil dieser Mutanten würde unerreichbare Knoten enthalten. Folglich wurde bei der Implementierung eine Überprüfung eingebaut, die solche isolierten Regelteile ausschließt. Da sich dadurch automatisch eine Reduktion der Anzahl entsprechender Mutanten einstellt, wird dieses Problem gleich mit entschärft.

RerouteOneLvEnd Beim Anlegen von *LVs* müssen Quell- und Zielknoten ausgewählt werden. Hierbei kann ein Entwickler selbstverständlich auch ein falsches Element auswählen. Gesetzt den Fall, dass als Ergebnis trotzdem eine Kante vom ursprünglich gewünschten Typ entsteht, würde der Irrtum unter Umständen nicht weiter auffallen. Folglich sollten auch solche Fehler durch einen Mutator simuliert werden. Der *RerouteOneLvEnd*-Operator tut genau dies, indem er für jedes Ende einer *LV* ein alternatives Ende innerhalb der umschließenden Regel sucht. Für jedes alternatives Endelement wird dann ein Mutant erzeugt, indem die ursprüngliche *LV* auf dieses neue Element „umgebogen“ wird. Eine konkrete Implementierung könnte zuvor auch noch überprüfen, dass beim „Umbiegen“ keine Situation entsteht, in der die Elemente der Regel in unerreichbare Komponenten zersplittern. Diese Aussage deutet aber bereits darauf hin, dass hier auf eine Umsetzung dieses Mutationsoperators verzichtet wurde.

RerouteBothLvEnds Analog zum zuvor betrachteten Operator ,liegt ein weiterer Mutationsoperator nahe, der nicht nur ein Ende einer *LV* neu verbindet, sonder beide Enden modifiziert. Hierbei ist allerdings fraglich, ob ein solcher Fehler in der Praxis realistischerweise tatsächlich auftritt.

AddRandomLv Im Gegensatz zum falschen Anlegen einer *LV* mit Bezug auf ihre beiden Endpunkte, erscheint es dagegen sehr viel wahrscheinlicher, dass eine überflüssige Kante im Musteranteil einer Regel vorkommt. Oft bestehen implizite Abhängigkeiten zwischen verschiedenen Referenzen eines Objekts oder mehrerer Objekte, so dass diverse funktional äquivalent Regelvarianten existieren können. Werden vermeintlich redundante Kanten im Muster angegeben, so besteht allerdings immer auch die Chance, dass durch die zusätzliche Einschränkung eigentlich erwünschte Treffer im Modell noch ausgeschlossen werden, beispielsweise weil ein Sonderfall übersehen wurde oder die implizite Annahme gar nicht gegeben ist. Entsprechend lassen sich Implementierungen dahingehend mutieren, dass zusätzlich *LVs* in die Implementierung eingebaut werden. Allerdings hat dieser Ansatz einige Nachteile, da er zum einen nicht sehr zielgerichtet erfolgen kann und tendenziell auch zu sehr vielen Mutanten führt. Darüber hinaus beseht auch eine nicht unerhebliche Gefahr dahingehend, dass Resultate äquivalente Mutanten darstellen.

Übersicht der Mutationsoperationen

In der nachfolgenden Tabelle 8.2 sind die beschriebenen Mutationsoperatoren noch einmal kompakt zusammengefasst. In der „Kontext“-Spalte ist verzeichnet, ob sich der Mutationsoperator auf den Kontrollfluss (CF) oder den Graphtransformationsanteil (GT) bezieht. Daneben ist verzeichnet, ob der Operator im Rahmen der Arbeit implementiert wurde (Spalte „Umgesetzt“) und auf welche Elemente, bezogen auf das *SDM*-Metamodell aus Anhang E, sich der Operator bezieht (Spalte „Bezugselement(e)“).

Nr.	Kontext	Name	Umgesetzt	Bezugselement(e)
1	CF	*AddStopNodeForUnguardedTransition*	✓	`StoryNode`, `StopNode`
2	CF	*SwitchSuccessAndFailureConditions*	✓	`ActivityEdge`
3	CF	*ConvertForEachToRegularPattern*	✓	`StoryNode`
4	CF	*ReturnDefaultValueOnStopNode*	✓	`StopNode`, `LiteralExpression`
5	CF	*ChangeStatementNodeParameterBinding*	✓	`StatementNode`
6	CF	*ReplaceParameterBindingByDummyLiteral*	✓	`StatementNode`, `LiteralExpression`
7	GT	*SubstituteTypeBySubtype*	✓	`ObjectVariable`
8	GT	*SubstituteTypeBySupertype*	✓	`ObjectVariable`
9	GT	*NegativeToOptional**	✓	`ObjectVariable`
10	GT	*NegativeToMandatory**	✓	`ObjectVariable`
11	GT	*OptionalToNegative**	✓	`ObjectVariable`
12	GT	*OptionalToMandatory**	✓	`ObjectVariable`
13	GT	*MandatoryToNegative**	✓	`ObjectVariable`
14	GT	*MandatoryToOptional**	✓	`ObjectVariable`
15	GT	*BoundToUnbound*†	✓	`ObjectVariable`
16	GT	*UnboundToBound*†	✗	`ObjectVariable`
17	GT	*CheckToCreate*‡	✓	`ObjectVariable`
18	GT	*CheckToDestroy*‡	✓	`ObjectVariable`
19	GT	*CreateToCheck*‡	✓	`ObjectVariable`
20	GT	*CreateToDestroy*‡	✓	`ObjectVariable`
21	GT	*DestroyToCreate*‡	✓	`ObjectVariable`
22	GT	*DestroyToCheck*‡	✓	`ObjectVariable`
23	GT	*RemoveAllAttributeConditions*§	✓	`ObjectVariable`, `Constraint`
24	GT	*RemoveOneAttributeCondition*§	✓	`ObjectVariable`, `Constraint`
25	GT	*AlterComparisonOperatorInAttributeCondition*§	✗	`ObjectVariable`, `Constraint`, `ComparisonExpression`
26	GT	*RemoveAllAssignments*¶	✓	`ObjectVariable`, `AttributeAssignment`
27	GT	*RemoveOneAssigment*¶	✗	`ObjectVariable`, `AttributeAssignment`
28	GT	*AssignDefaultValue*¶	✗	`ObjectVariable`, `AttributeAssignment`, `LiteralExpression`
29	GT	*BindingExpression*-Mutationen	✗	`ObjectVariable`
30	GT	*RemoveOv*	✗	`StoryPattern`, `ObjectVariable`
31	GT	*AddUnconnectedOv*	✗	`StoryPattern`, `ObjectVariable`
32	GT	*AddConnectedOv*	✗	`StoryPattern`, `ObjectVariable`, (`LinkVariable`)
33	GT	*SubstituteLvType*	✓	`LinkVariable`, (`ObjectVariable`)
34	GT	*ReverseLvDirection*	✗	`LinkVariable`, (`ObjectVariable`)

* *AlterOvBindingSemantic*
† *AlterOvBindingState*
‡ *AlterOvBindingOperator*
§ *AttributeCondition*
¶ *AttributeCondition*

Nr.	Kontext	Name	Umgesetzt	Bezugselement(e)
35	GT	*RemoveLv*	✓	`StoryPattern`, `LinkVariable`, (`ObjectVariable`)
36	GT	*RerouteOneLvEnd*	✗	`StoryPattern`, `LinkVariable`, (`ObjectVariable`)
36	GT	*RerouteOneLvEnd*	✗	`StoryPattern`, `LinkVariable`, (`ObjectVariable`)
37	GT	*RerouteBothLvEnds*	✗	`StoryPattern`, `LinkVariable`, (`ObjectVariable`)
38	GT	*AddRandomLv*	✗	`StoryPattern`, `LinkVariable`, (`ObjectVariable`)

Tabelle 8.2: Übersicht der Mutationsoperatoren (CF: Control-Flow, GT: Graphtransformation)

8.4 Implementierung

Für eine praktikable Anwendung des mutationsbasierten Testens ist ein hohes Maß an Automatisierung erforderlich, wie das folgende Zitat aus [AO08, S. 273] unterstreicht:

> „It [*mutation, Anmerkung des Autors*] is also all but impossible to apply by hand; thus automation is a must. Not surprisingly, automating mutation is more complicated than automating other criteria."

Aus diesem Grund wurden im Rahmen dieser Arbeit die wichtigsten der vorgestellten Mutationsoperatoren als Teil eines Mutationsrahmenwerks für *SDM*-Transformationen implementiert. Die Umsetzung erfolgte als Teil der eMoflon-Tool-Suite auf Eclipse-Basis, wobei die Implementierung ein (Haupt-)Plugin umfasst, welches den Anwender bei der Durchführung der folgenden zentralen Aufgaben unterstützt:

1. Ableitung von Mutanten inklusive der automatisierten Codegenerierung für diese.
2. Automatisierte Auswertung und Evaluation einer Testmenge auf Grundlage der mutierten Transformationsvarianten (inklusive Protokollierung der Ergebnisse).

Im Folgenden wird zunächst die grundlegende Funktionalität beschrieben, insbesondere aus Sicht eines Anwenders. Danach werden die wesentlichen technischen Konzepte der Umsetzung vorgestellt, bevor die Darstellung der Implementierung mit einer kurzen Analyse von potentiellen Optimierungsmöglichkeiten abgeschlossen wird.

8.4.1 Das `SdmMutationFramework`-Plugin

Die Umsetzung des Mutationsrahmenwerks erfolgte als Eclipse-Plugin, welches die Bezeichnung „`SdmMutationFramework`" trägt. Es definiert Dialoge, Aktionen und sonstige Erweiterungen der IDE zur Durchführung der Mutationsanalyse bei *SDM*-Transformationen und baut dazu auf den Plugins der eMoflon-Tool-Suite auf, welche an entsprechenden Stellen erweitert werden. Der überwiegende Teil der Funktionalität bezieht sich

auf die Erzeugung der Mutanten. Ein Prozess, der, wie im Folgenden erläutert, selbst als Modelltransformationsproblem angesehen werden kann.

Zur besseren Darstellung werden zuerst die wesentlichen Arbeitsschritte, die das Rahmenwerk ermöglicht, vorgestellt. Danach wird die eigentliche Realisierung skizziert.

Anwendungsfälle und Kernfunktionen

Wie bereits erwähnt, dient das Plugin zweier Aufgaben. Zur Umsetzung müssen diese allerdings noch enger gefasst werden. Hierzu dienen zwei Use-Case-Diagramme, welche die wesentlichen Aktivitäten für beide Aufgaben konkretisieren.

Abbildung 8.8 stellt ein Use-Case-Diagramm zur eigentlichen Erzeugung der Mutanten dar. Als Akteure sind der Tester sowie das Mutationswerkzeug, hier kurz als „System“ bezeichnet, involviert. Der Hauptanwendungsfall[8] aus Sicht des Testers besteht im Generieren der Mutanten. Dies beinhaltete das Selektieren eines bestehenden *SDM*-Transformationsprojektes im aktuellen Eclipse-Workspace sowie das Auswählen der konkreten Mutationsschritte aus der Menge aller möglichen Optionen. Letztere werden durch das System mit Hilfe der realisierten Mutationsoperatoren und in Abhängigkeit von der konkreten *SDM*-Spezifikation bestimmt. Dieser Anwendungsfall, und einer der beiden Hauptanwendungsfälle aus Sicht des Systems, umfasst das Bestimmen aller möglichen Anwendungsstellen der jeweiligen Mutationsoperatoren auf Grundlage einer Analyse der Transformation. Außerdem werden dem Tester die identifizierten konkreten Mutationsaktionen vom System zur Auswahl präsentiert. Der Tester wählt einige (oder alle) Aktionen aus, die er tatsächlich durchgeführt haben möchte und lässt diese durch das System ausführen. Das System wertet hierzu die Wahl des Testers aus und erzeugt für jede durchzuführende Aktion/Mutation eine Kopie des ursprünglichen *SDM*-Diagramms. Außerdem stellt das System dabei sicher, dass die Entsprechungen der identifizierten und referenzierten Entitäten aus der ursprünglichen Transformation in der Kopie der Transformation gefunden werden, so dass diese für die weiteren Schritte verwendet werden. Das ist notwendig, damit sich alle durchgeführten Mutationen auch auf die aktuellen Kopien beziehen und die Transformationsbeschreibung für die Durchführung weiterer Mutationen erhalten bleibt. Zur weiteren Verarbeitung, insbesondere zur Codegenerierung, müssen die mutierten Diagramme dann noch vom System serialisiert werden.

In Abbildung 8.9 ist ein zweites Use-Case-Diagramm angegeben, das die Bewertung einer Testmenge auf Basis der erzeugten Mutanten thematisiert. Der Hauptanwendungsfall aus Benutzersicht besteht in der Bewertung einer zu erstellenden Testmenge,[9] welcher hier als konkreter Anwendungsfall des abstrakten Anwendungsfalls „Mutanten nutzen“ modelliert wurde. Letzterer ist deshalb vorhanden, da die erzeugten Mutanten grundsätzlich auch zur *Neuentwicklung* von Tests genutzt werden können. Ein entsprechender Anwendungsfall wurde in der Darstellung allerdings bewusst ausgespart. Das Bewerten der Test-Suite umfasst das Erstellen der Testmenge sowie das Starten der Bewertung. Das System führt die Bewertung selbstständig durch, indem es einzelne Mutanten auswählt und aktiviert sowie die gegebene Testmenge auf der mutierten *SDM*-Transformation zur Ausführung bringt. Sollte für die mutierte *SDM*-Spezifikation noch keine ausführbare

[8] Primäre Anwendungsfälle sind hier durch schwarze Kanten mit dem jeweiligen Akteur verbunden, sekundäre dagegen durch graue Kanten.

[9] Hiermit ist nicht notwendigerweise gemeint, dass die Testmenge neu erstellt werden muss, sondern, dass die Menge der auszuführenden Tests konfiguriert wird.

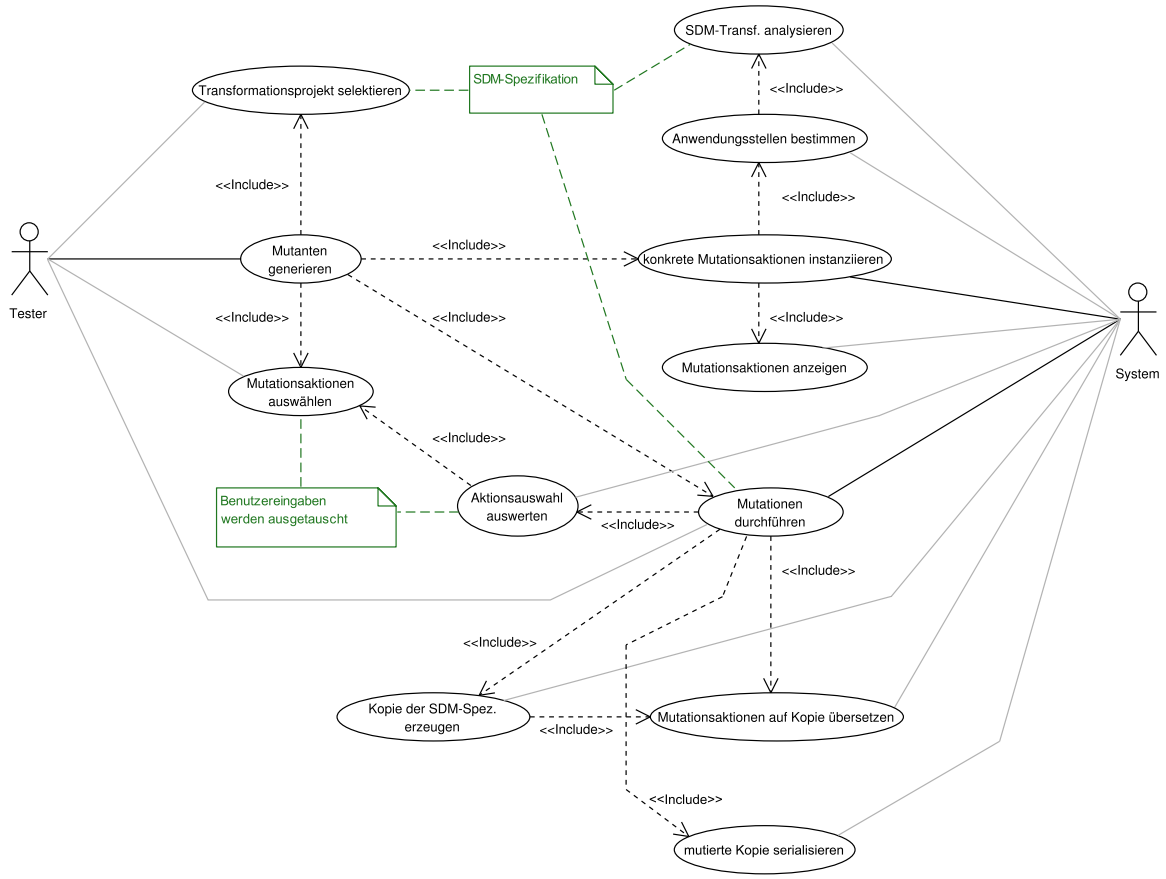


Abbildung 8.8: Use-Case-Diagramm zur Erzeugung der Mutanten

Repräsentation – in unserem Fall in Form von Java-Code – vorliegen, so muss diese vor der Ausführung noch erzeugt werden. Im Idealfall erfolgt auch dieser Schritt automatisiert, was hier auch geschieht. Das Ausführen beinhaltet als einen integralen Bestandteil das Auswerten und Protokollieren der Ergebnisse der Testausführung.

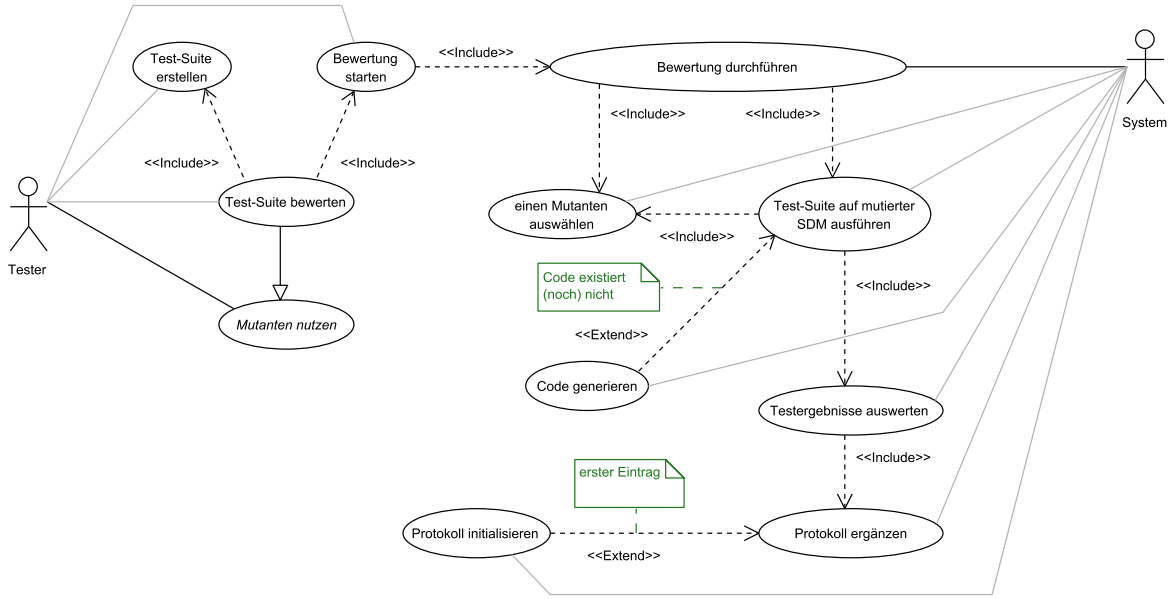


Abbildung 8.9: Use-Case zur Bewertung einer Test-Suite

Zur Umsetzung

In Abbildung 8.10 sind die zur Ausführung einer Mutationsanalyse notwendigen Schritte skizziert. Ausgangspunkt ist ein mittels Tests zu verifizierendes Transformationsprojekt – in der Abbildung konkret am Beispiel des BD2Ja-Projekts – das die *SDM*-Diagrammbeschreibungen umfasst sowie gegebenenfalls weitere benötigte Metamodellprojekte referenziert – hier `JavaLanguage` und `BlockDiagramLanguage`. In der Abbildung ist im oberen linken Bereich eine entsprechende Sicht auf den Eclipse-Workspace, der das Transformationsprojekt mit dem unveränderten (Meta-)Modell inklusive der Transformationsdefinition beinhaltet, angedeutet. Durch einen Rechtsklick auf das Transformationsprojekt erreicht man ein kontextsensitives Menü und darüber die Funktionalität des Mutationsrahmenwerks, in dem man die durchzuführende Aktion auswählt. In der Abbildung 8.10 sind folgende Teilschritte angegeben:

1. Test-Suite-Projekt anlegen: Per „Create Testsuite“ wird ein dediziertes Testprojekt neu angelegt (hier `BD2JaTestSuite`). Innerhalb dessen können JUnit-Tests von Grund auf neu erstellt werden, vgl. Schritt 5, oder eine zu bewertenden Test-Suite importiert bzw. diese referenziert werden. Da sich die Ausführung der Tests aufgrund der zu automatisierenden wiederholten Auswertung auf jedem Mutanten von einer Standardauswertung einer Testmenge unterscheidet, wurde ein neuer Eclipse-Projekttyp für ein solches Projekt eingeführt. Darüber hinaus dient das Projekt als Speicherort der anzulegenden Testprotokolle.

2. Mutantenerzeugung starten: Durch die Auswahl der zweiten Option im Kontextmenü des ursprünglichen Transformationsprojektes wird die Transformation analysiert, so dass Anwendungsstellen der verschiedenen Mutationsoperatoren bestimmt werden können. Die resultierenden Optionen werden anschließend dem Benutzer zur Auswahl präsentiert, vgl. Schritt 3.

3. Zu erzeugende Mutanten auswählen: Der Benutzer wählt mit Hilfe der erzeugten Auswahlliste, die alle Optionen umfasst, eine Menge konkret zu erzeugender Mutanten aus. Im Anschluss daran ist noch die Erzeugung eines entsprechenden Mutationsprojekts (im konkreten Fall das `BD2JaMutation`-Projekt) zu bestätigen. Dieses wird vom ursprünglichen Transformationsprojekt durch Kopieren abgeleitet, um das ursprüngliche Projekt nicht zu modifizieren.

4. Code für die mutierten Transformationsvarianten generieren: Das Mutationsprojekt enthält neben dem ursprünglichen (Meta-)Modell (`BD2Ja.ecore`) auch alle serialisierten Mutanten (`.ecore.mut`-Dateien). Standardmäßig ist Eclipse so konfiguriert, dass unmittelbar nach der Erzeugung des Projektes der automatische Build-Prozess für das Projekt einsetzt. Sollte das automatische Bauen von Projekten global deaktiviert sein, so muss der Benutzer den Build-Prozess manuell starten. Der Build-Prozess stellt dann sicher, dass für *alle* mutierten Transformationsvarianten Code erzeugt wird. Da anschließend zu jeder Variante ein vollständiger Satz an Quellcodedateien vorliegt, kann eine der Varianten leicht durch Anpassen des Java-Build-Pfades und *ohne* erneutes Generieren des Codes aktiviert werden. Nach der Codegenerierung ist das Mutationsprojekt bereit zur Verwendung.

5. Tests schreiben oder bereitstellen: Auf technischer Ebene werden Tests als JUnit-Testfälle formuliert. Wie bereits angedeutet, müssen die Testfälle für den hier vorgeschlagenen Anwendungsfall „Bewertung einer Testmenge“ bereits vorhanden sein. Allerdings spricht grundsätzlich auch nichts dagegen, die Tests innerhalb des Test-Suite-Projektes komplett neu zu entwickeln.

6. Run-Configuration erstellen und auszuführende Tests auswählen: Vergleichbar zum Anlegen einer Konfiguration für normale JUnit-Tests, muss auch für die Ausführung der Tests im Falle der Berechnung des Mutationswertes eine sog. Run-Configuration angelegt werden, die beispielsweise festlegt, welche Tests ausgeführt werden sollen und wie die dazu notwendige Java-Laufzeitumgebung zu konfigurieren ist. Die Besonderheit hier besteht darin, dass das Plugin des Mutationsrahmenwerks einen eigenen Typ von Run-Configuration einführt, auf der eine angepasste `ILaunchConfigurationDelegate`[10]-Instanz aufbaut, die sequenziell die Tests auf allen Mutanten zur Ausführung bringt.

7. Test-Suite ausführen: Die Testausführung startet durch Ausführen der neu angelegten Run-Configuration. Hierbei wird programmatisch der Build-Pfad des Java-Codes sukzessive so angepasst, dass genau ein Mutant aktiv ist. Anschließend wird der Java-Compiler, aber *nicht* der Codegenerator, aktiv. Die Tests werden nach dessen Lauf in einer zusätzlichen *JVM*-Instanz ausgeführt und das Ergebnis der

[10] Vergleiche `org.eclipse.debug.core.model.ILaunchConfigurationDelegate` der Eclipse-Plattform, Version 4.3.1.

Testläufe, nicht der Überdeckung, wird in serialisierter Form über das Dateisystem gesammelt zurückgeliefert.

8. Report generieren: Nachdem die Tests auf allen Mutanten ausgeführt wurden, werden danach die serialisierten Ergebnisse aller Läufe gesammelt eingelesen und daraus ein abschließender Bericht generiert. Hierbei sind *fehlgeschlagene*, rote Tests grundsätzlich als positiv anzusehen, da sie den jeweiligen Mutanten aufdecken.[11]

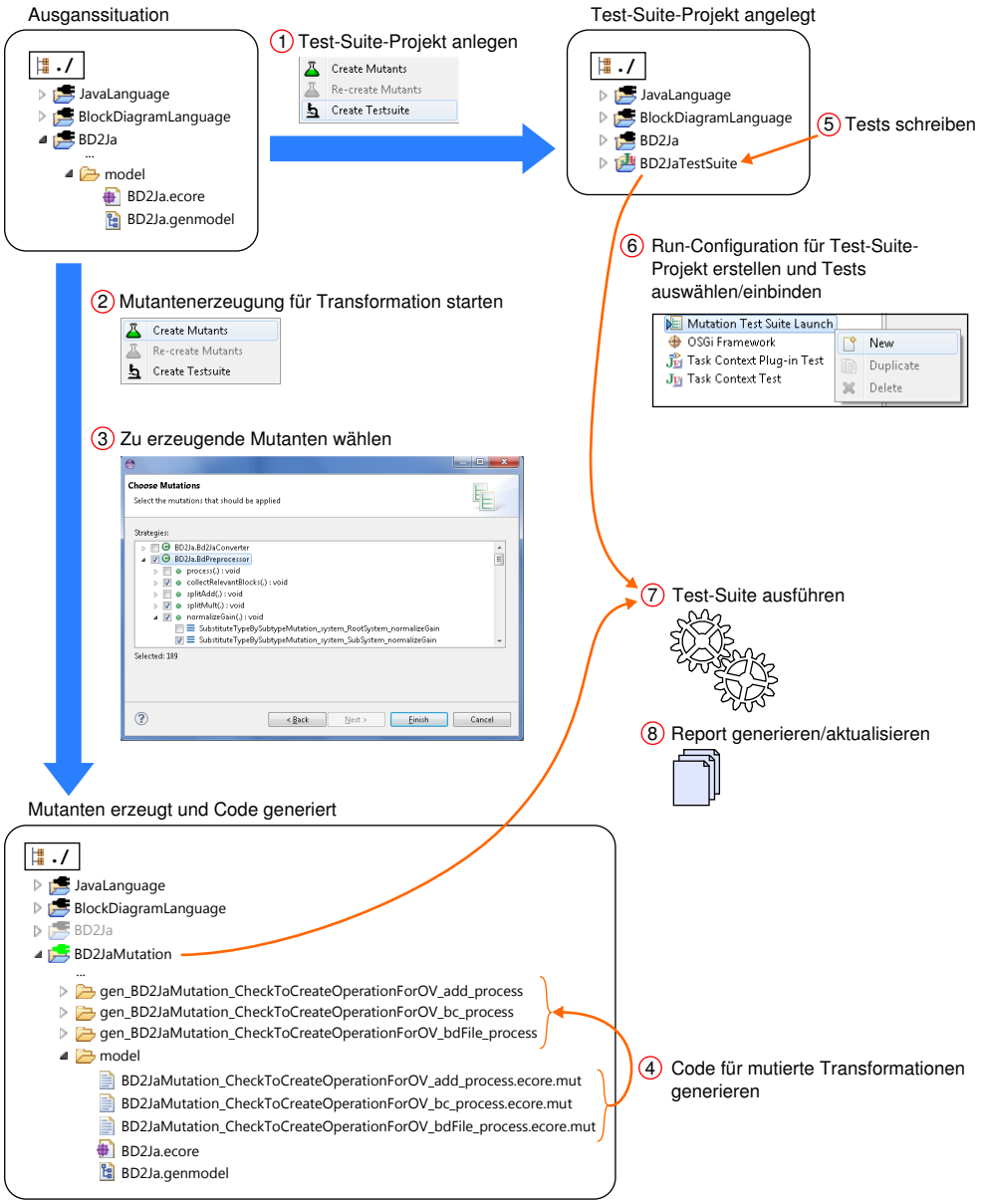


Abbildung 8.10: Das Mutationsrahmenwerk aus Anwendersicht

[11] Dies setzt voraus, dass das unmutierte System fehlerfrei im Sinne der Testmenge ist. Auch stimmt dies nur näherungsweise, da grundsätzlich auch falsche Positiv-Erkennungen möglich sind, wenn auch in diesem Sinne fehlgeschlagene Tests eigentlich zu vermeiden sind.

8.4.2 Technische Besonderheiten

Im Rahmen der Entwicklung des Plugins mussten einige Herausforderungen technischer Art gelöst werden. Hier werden die wichtigsten Probleme und Lösungen kurz vorgestellt, die spezifisch für die Mutantenerzeugung aber unabhängig von den konkret umgesetzten Mutatoren sind.

Mutantenerzeugung mittels Higher-Order-Transformationen

Beim Mutieren von *SDM*-Transformationen stellt sich die Frage, wie die Mutationsschritte programmatisch umzusetzen sind. Da die Transformationsbeschreibung als Ecore-Modell vorliegt, liegt eine Java-Implementierung unter Nutzung der *EMF-API* nahe. Standen allerdings bei der zeitlich früher stattgefundenen Implementierung des *RPC*-Ansatzes noch algorithmische Probleme im Rahmen komplexerer Analysen im Vordergrund – diese ließen sich äußerst flexibel mit Hilfe von Java entwickeln und nachträglich anpassen – steht bei der Mutation vor allem die Suche nach Anwendungsstellen für die unterschiedlichen Operatoren im Vordergrund. Dies stellt eine Paradedisziplin für Sprachen wie *SDM* dar, in denen die Suche von (Graph-)Mustern bereits fester Bestandteil ist. Insbesondere, da die zu mutierende Transformationsbeschreibung bereits einem bekannten Metamodell genügt, nämlich dem *SDM-Metamodell (MM)*, und die Mutatoren bereits mit Bezug zu diesem beschrieben wurden, liegt eine Implementierung der *SDM*-Mutierung als *SDM*-Transformation in Form einer *Higher-Order Transformation (HOT)* bzw. *Meta-Transformation*, vgl. [VP04], nahe. Für eine Übersicht typischer *HOT*-Szenarien siehe beispielsweise [Tis+09]. Dort ist insbesondere der Abschnitt 7.1 zu „Transformation Variants“ in Bezug auf den hier vorliegenden Anwendungsfall relevant.

Die zuvor beschriebenen Mutatoren wurden im Rahmen der Arbeit in einem eigenen Metamodell, das die wesentlichen Konzepte der Struktur des Mutationsrahmenwerks definiert, erfasst. Die eigentlichen Mutationsschritte bilden mehrstufig ablaufende Transformationen, spezifiziert in der *SDM*-Sprache. Diese manipulieren selbst wiederum *SDM-Transformationen* (inklusive der referenzierten Instanzen der zugehörigen Sprachdefinition(en)) als Ein- und Ausgabemodelle. Das Vorgehen führ dazu, dass grundsätzlich die Möglichkeit besteht, die Implementierung der Mutationoperatoren selbst durch mutationsbasiertes bzw. *RPC*-basiertes Testen abzusichern. Leider war dies im Rahmen der Arbeit zeitlich nicht mehr möglich, da aus den verschiedensten Gründen, auf die im nächsten Kapitel noch genauer eingegangen werden soll, eine andere umfangreiche Transformation im Rahmen der Evaluation zugrundegelegt wurde. Didaktische Gründe sprachen ebenfalls gegen die Verwendung dieser *HOT* als Beispiel im Rahmen dieses Arbeit, da hierbei Bezeichner, Namen und Konzepte in unterschiedlichen Rollen vorgekommen wären, was das Verständnis zusätzlich erschwert hätte.

Drei Schritte: Suchen und Identifizieren, Klonen, Wiederfinden

Die Ableitung der Mutanten erfolgt schrittweise. Zuerst werden entsprechende Anwendungsstellen für die einzelnen Mutationoperatoren gesucht. Dann werden Referenzen auf die von Änderungen betroffenen Elemente bis zur eigentlichen Ausführung der Manipulation abgespeichert. Da allerdings nur eine einzelne Änderung pro Mutant zulässig ist, muss die in den Arbeitsspeicher geladene ursprüngliche Transformationsbeschreibung vor der Durchführung der Änderung kopiert bzw. geklont werden. Ansonsten müsste die

Originalbeschreibung vor jeder Manipulation erst deserialisiert und erneut in den Speicher eingelesen werden. Dies hätte allerdings Nachteile aufgrund der wesentlich größeren Anzahl an langsamen Festspeicherzugriffen im Vergleich zu dem hier verfolgten Ansatz, bei dem das Originalmodell nur einmalig geladen und analysiert wird und nur Kopien manipuliert und serialisiert werden.

Als problematisch bei der hier vertretenen Implementierungsvariante stellt sich allerdings die Tatsache heraus, dass sich die zuvor erwähnten Referenzen, welche die Anwendungsstellen der Manipulation identifizieren, ohne weitere Anpassungen auf die Elemente des ursprünglichen Modells beziehen. Damit bei der Durchführung einer Mutation das ursprüngliche Modell unverändert bleibt und so unbeteiligte Referenzen ihre grundsätzliche Gültigkeit behalten, müssen zuvor die Entsprechungen der von der Mutation betroffenen Elemente in der Kopie gesucht und wiedergefunden werden. Aufgrund der Tatsache, das Kopie und Original strukturell übereinstimmen, ist das Wiederfinden relativ leicht umzusetzen, selbst wenn dieser Schritt in der *SDM*-Sprache implementiert wird.[12]

Automatisierte Testausführung und Report-Generierung

Zur Automatisierung der Testausführung wurde, wie weiter oben bereits erwähnt, eine eigene Implementierung der `ILaunchConfigurationDelegate`-Schnittstelle realisiert. Diese passt den Build-Pfad an und erzeugt einen neuen sog. *Task* zur Ausführung der JUnit-Tests. Letzterer basiert auf der Implementierung eines JUnit-Tasks[13] des *Apache-Ant*-Rahmenwerks. Mittels eigener Unterklasse von *org.apache.tools.ant.BuildListener* wird die Anzahl der fehlgeschlagenen sowie der ausgeführten Tests erfasst, so dass der Mutationskoeffizient berechnet werden kann.

Ohne ein aussagekräftiges und nachvollziehbares Testprotokoll ist die Ausführung der Tests auf den Mutanten allerdings nur von begrenztem Wert, da ansonsten die notwendige Nachvollziehbarkeit nicht gegeben ist. Glücklicherweise bietet der zuvor erwähnte Ant-Task eine entsprechende Option zur Protokollierung der einzelnen Testläufe an. Hierzu werden mittels *FormatterElement*-Instanzen,[14] welche auf die Formate „plain“ (einfache Textzusammenfassung) und „xml“ (xml-Format der JUnit-ResultView in Eclipse) eingestellt sind, Ausgaben für jeden Testlauf im Dateisystem abgelegt. Sind alle Tests auf allen Mutanten ausgeführt, wird anschließend mittels `XMLResultAggregator`-Instanz[15] ein aggregierter Bericht aus den Einzeldateien abgeleitet, wiederum einmal als xml-Version (Option „xml“), der kompatibel zur JUnit-ResultView von Eclipse ist, und einmal als anwenderfreundlicher HTML-Bericht (Option „frames“).

8.4.3 Optimierungsmöglichkeiten

Die entstandene Implementierung zur Mutantenerzeugung, Testausführung und Reporting ist in vielerlei Hinsicht als prototypisch zu betrachten. Bei der Entwicklung galten die Devisen „Funktionalität vor Effizienz“ und „Wiederverwendung vor Neuentwicklung“. Letzteres unter möglichst ausgiebiger Nutzung existierender und getesteter Werkzeuge

[12] Für die ausschlaggebende Entscheidung bezüglich der Objektidentität einzelner Elemente wird hierzu allerdings an *EMF*-Code delegiert.

[13] `org.apache.tools.ant.taskdefs.optional.junit.JUnitTask`

[14] `org.apache.tools.ant.taskdefs.optional.junit.FormatterElement`

[15] `org.apache.tools.ant.taskdefs.optional.junit.XMLResultAggregator`

sowie Plugins, wie z. B. von eMoflon oder der *EMF*- und *Java Development Tools (JDT)*-Funktionen in Eclipse. Darüber hinaus ist der Mutationsprozess und die Umsetzung der Mutationsanalyse an sich stellenweise suboptimal umgesetzt, da beispielsweise für jeden Mutanten stets eine weitestgehend identische Kopie der gesamten Spezifikation (Metamodell plus Transformationsbeschreibungen) erstellt wird bzw. wirklich jeder Testfall auf jedem Mutanten ausgeführt wird. Im Anschluss werden deshalb einige Möglichkeiten für eine zukünftige Optimierung beschrieben.

Caching des Generats

Nach der Erzeugung eines Mutanten muss für die nun veränderte Transformationsbeschreibung Code generiert werden. Entscheidend ist dabei, dass dieser Prozess nur einmalig erfolgt und nicht bei jeder Testausführung erneut durchlaufen wird. Letzteres ist hier bereits nicht der Fall, da die kompletten Codestände aller Mutanten separat im Dateisystem abgelegt werden. Dieses Vorgehen bietet allerdings noch Raum für Optimierungen. So entstehen für größere Transformationen sehr viele Quellcode-Dateien. Deren Inhalt ist im direkten Vergleich zwischen einzelnen Varianten überwiegend identisch.

Eine naheliegende Optimierung hinsichtlich reduziertem Speicherplatzbedarf bestünde darin, die generierten Codeverzeichnisse zu komprimieren; eine gute Komprimierbarkeit ist zudem gegeben. Ein Nachteil einer solchen Lösung besteht in einem Anstieg der Laufzeit der Mutationsanalyse aufgrund der notwendigen Dekompression. Andererseits wäre es auch möglich, pro Mutant nur die tatsächlich veränderten Codefragmente zu identifizieren und ausschließlich entsprechende Deltas abzuspeichern und zwischen den Testläufen auszutauschen. Dazu wären allerdings entweder relativ umfangreiche Vergleiche der Codestände notwendig – der eingesetzte Codegenerator erzeugt bei mehreren Läufen zum Teil unterschiedliche aber äquivalente Varianten – oder aber die Mutationskomponente müsste unmittelbare Hinweise geben, an welchen Stellen mit Unterschieden zu rechnen wäre.

Eine weitere Optimierung, die beispielsweise bei der Mutation im Zusammenspiel mit klassischen Programmiersprachen häufiger zum Einsatz kommt, vgl. z. B. [AO08, Abschn. 8.2.3, S. 276 f.], liegt in der Kombination aller Mutanten in *einer gemeinsamen* Beschreibung. Ein einzelner Mutant wird dabei entweder durch Konfiguration dynamisch zur Laufzeit anhand auszuwertender Bedingungen ausgewählt – z. B. mittels `switch`- oder `if`-Anweisungen) – oder durch statisch auszuwertende Präprozessoranweisungen nur bedingt erzeugt. Die erste Variante böte den Vorteil, dass der sog. *Compilation Bottleneck*, der durch mehrfaches Aufrufen des Compilers entsteht, vgl. [AO08, S. 275], durch das einmalige Kompilieren weitestgehend entfallen würde. Für die zweite Variante spricht dagegen, dass die jeweilige Ausführungszeit durch Verzicht auf die Fallunterscheidungen kürzer ist.

Caching des Kompilats

Im letzten Absatz sollte bereits deutlich geworden sein, dass ein erneutes Kompilieren der Codestände die Ausführungszeit bei der Mutationsanalyse negativ beeinflussen kann. Für den konkreten Fall bedeutet das, dass die Java-Class-Dateien nicht immer wieder neu erzeugt werden sollten, wenn das Codeverzeichnis auf eine andere Stelle eines neuen Mutanten „umgebogen“ wird. Neben dem Sichern des aus dem mutierten Modell abgeleiteten

Codes könnte man folglich auch den aus dem Quellcode mit Hilfe des Java-Compilers erzeugten Bytecode für jeden Mutanten sichern. Somit müsste der Java-Compiler nicht bei jedem Ausführen der Tests wiederholt den Quellcode kompilieren. Hierbei handelt es sich ebenfalls um ein Beispiel für den klassischen Zielkonflikt zwischen größerem Speicherplatzbedarf und besserer Laufzeitperformanz.

Selektives Ausführen der Tests

Das Ziel einer Mutationsanalyse liegt in der Analyse, ob alle bzw. wie viele der Mutanten durch Tests identifiziert werden. Prinzipiell reicht für das Identifizieren pro Mutant bereits ein einzelner fehlgeschlagener bzw. roter Test. Fortgesetztes Testen eines bereits getöteten Mutanten führt zu keinem weiteren Gewinn an Erkenntnis im Hinblick auf das angestrebte Ziel. Dennoch kann sich auch das fortgesetzte Testen als sinnvoll erweisen, da man hierdurch ggf. Erkenntnisse über die Zusammensetzung der Testmenge erhalten kann und eventuell überflüssige Tests herausstechen. Unabhängig von diesem letzten Punkt wäre es allerdings wünschenswert, dass ein Ausführungsmodus vorhanden ist, in dem das Testen nach dem ersten fehlgeschlagenen Test mit der Überprüfung auf Basis des nächsten Mutanten fortfährt. In der aktuellen Implementierung fehlt eine solcher Modus leider noch.

Wäre der besagt Modus allerdings verfügbar, könnten bei Bedarf zusätzlich die Testfälle noch (automatisiert) analysiert und in eine solche Reihenfolge gebracht werden, die ein möglichst frühes Fehlschlagen wahrscheinlicher erscheinen lassen. Z. B. könnten für einen bestimmten Mutanten solche Testfälle bevorzugt ausgeführt werden, die insbesondere Codebereiche exerzieren, die dadurch von der Mutation betroffen sind. Diese Idee könnte im Rahmen weitergehender Untersuchungen betrachtet werden.

8.5 Anwendung

Im Rahmen initialer Versuche, insbesondere bei der Anwendung des Mutationsrahmenwerks auf die *SDM*-Beispieltransformation aus Abschnitt 4.4, zeigte sich die grundsätzliche Anwendbarkeit und Funktionsfähigkeit des Ansatzes. Auf die Darstellung konkreter Ergebnisse und Statistiken soll an dieser Stelle jedoch verzichtet werden. Das nachfolgende Kapitel 9, insbesondere der Unterabschnitt 9.2.2, geht speziell auf die entsprechenden Ergebnisse im Rahmen der Anwendung im Kontext einer noch umfangreicheren *MT* ein.

8.6 Bewertung

Widmen wir uns zum Abschluss des Kapitels nun noch kurz einigen verallgemeinerbaren Ergebnissen und Erkenntnissen aus der Anwendung des Mutationsrahmenwerks.

- Die Machbarkeit bzw. Praktikabilität der Mutationen von *SDM*-Transformationen mit den Mitteln einer *HOT* konnte exemplarisch gezeigt werden.
- Das Ergänzen zusätzlicher Mutationstrategien ist durch entsprechende Ergänzungen im Metamodell und durch die Implementierung einiger weniger Operationen anhand von *SDM*-Beschreibungen problemlos möglich.
- Selbst mit der hier realisierten, überschaubaren Menge an konkreten Mutatoren entstehen schnell hunderte bis tausende potentielle Mutanten. Eine Automatisierung,

insb. der Testausführung, ist somit in Anlehnung an eine entsprechende Aussage zum Bau von Mutationswerkzeugen von Ammann und Offutt, die weiter vorne im Text bereits zitiert wurde, eine praktische Notwendigkeit.

- Die Dauer der Suche nach den Anwendungsstellen für alle Mutatoren ist selbst für mittelgroße – vergleichbar zu dem Beispiel aus Abschnitt 4.4 – oder noch umfangreichere (vgl. Kap. 9) Transformationen mit einigen wenigen Sekunden zwar wahrnehmbar, darf aber als vernachlässigbar gering angesehen werden.
- Die Dauer für die eigentliche Ausführung aller möglichen Mutationen und die Serialisierung der resultierenden Modelle ist mit 2..3 min für ca. 3000 Mutanten tolerabel. Zur Erinnerung, dieser Schritt ist nur relativ selten durchzuführen und lässt sich durch Beschränkung auf eine zufällige Teilmenge reduzieren.
- Der Aufwand des mutationsbasierten Testens erscheint dennoch sehr groß, da die Zeiten für Codegenerierung und Testausführung erheblich sein können, wie im nächsten Kapitel dargelegt. Insbesondere erscheint das Verfahren für ein kontinuierliches, wiederholtes Testen während der Transformationsenwicklung eher ungeeignet, da nach Änderungen an der Transformationsimplementierung neue Sets an Mutanten erzeugt werden sollten.
- Die Automatismen zur Ausführung einer gegeben Testmenge auf allen generierten Mutanten haben in der praktischen Anwendung ihre Tauglichkeit gezeigt.
- Die generierten Testprotokolle sind relativ umfangreich und reichen mehr als aus, um die Mutationsüberdeckung bestimmen zu können.
- Erfahrungsgemäß sind nichtkompilierende Mutanten nicht vollständig auszuschließen. Sie können allerdings anhand der Testprotokolle identifiziert und von einer Berücksichtigung in Bezug auf die Mutationsadäquatheit ausgeklammert werden.
- Für die systematische Erzeugung einer großen Anzahl an fehlerhaften Varianten einer *SDM*-basierten *GT*, wie sie eine Beurteilung der Leistungsfähigkeit des *RPC*-Ansatzes erfordert, erscheint das Mutationsrahmenwerk grundsätzlich geeignet zu sein. So lassen sich ausreichend viele Mutanten erzeugen und eine bestehende Testmenge leicht auf diesen automatisiert ausführen und bewerten. Darüber hinaus wurden beide Teilaspekte weitestgehend unabhängig voneinander realisiert, so dass eventuell vorhandene Implementierungsfehler sich nicht unmittelbar fortpflanzen konnten und so die Vertrauenswürdigkeit der Ergebnisse nicht von vorne herein leidet.
- Eine Abschätzung der allgemeinen Leistungsfähigkeit der durch diese Verfahren ableitbaren Testmenge muss an dieser Stelle noch offen bleiben. Für weitere Untersuchungen sei auf die Evaluation im dritten Teil der Arbeit verwiesen.

Teil III

Evaluation

Miss alles, was sich messen lässt, und mach alles messbar, was sich nicht messen lässt.

(Als Urheber werden sowohl Archimedes als auch Galileo Galilei kolportiert)

9 Experimente und Mutationsanalyse

Nachdem die Hauptbeiträge dieser Arbeit in den vorangegangenen Kapiteln ausführlich beschrieben worden sind, soll es in diesem letzten Teil der Arbeit darum gehen, wie sich der entwickelte Testansatz auf Basis der *RP*-Überdeckung in der Praxis bewährt. Hierzu wurde der Ansatz im Kontext einer umfangreichen und komplexen *SDM*-Transformation, die nicht mit dem bisherigen Beispiel aus dieser Arbeit übereinstimmt, exemplarisch und praktisch erprobt. Die Transformation, die diesem Vorgehen zugrunde liegt, wurde zur Wahrung der größtmöglichen Objektivität der Resultate so ausgewählt, dass sie unabhängig von dieser Arbeit entwickelt wurde. Dazu gehört auch die Vermeidung personeller Überschneidungen bezüglich der Autorenschaft. Eine entsprechende Transformation ausreichender Größe – inklusive einer initialen Testmenge – konnte glücklicherweise dem wachsenden Fundus an Transformationsbeispielen des Instituts entnommen werden, so dass entsprechende Experimente darauf aufbauend durchgeführt werden konnten.

Der Vorstellung der ausgewählten konkreten Transformation ist der erste Unterabschnitt 9.1 dieses Kapitels gewidmet. Darüber hinaus sind verschiedene Fragestellungen zur Bewertung des Testansatzes von grundsätzlichem Interesse, deren Untersuchungen den restlichen Teil des Kapitels ausmachen. So wird in Abschnitt 9.2 die praktische Einsetzbarkeit der Implementierung betrachtet und dabei die Frage untersucht, ob die realisierte Funktionalität sowohl der *RPC*-Maschinerie als auch des Mutationsansatzes geeignet ist, mit der umfangreichen Transformation umzugehen. Darauf folgt in Abschnitt 9.3 eine Analyse der praktischen Durchführbarkeit des Testvorhabens auf Grundlage der *RPC* für das konkrete Beispiel. Dabei liegt ein Schwerpunkt einerseits auf der Identifikation von ggf. verallgemeinerbaren praktischen Erfahrungen und andererseits auf darauf aufbauenden Hypothesen, bezogen auf die Nützlichkeit des Verfahrens bei der manuellen Ableitung guter und sinnvoller Tests. In Abschnitt 9.4 wird der Frage nachgegangen, ob und mit welchem Ergebnis – über eine Fehlererkennungsrate der konkreten, anhand einer Steigerung der *RP*-Überdeckung verbesserten Testmenge, bestimmt durch Mutationsanalysen – auf die Leistungsfähigkeit des Verfahrens im Allgemeinen geschlossen

werden kann. Insbesondere dieser letzte Teil ist als initiale, nicht abgeschlossene Untersuchung und als Versuch einer initialen Einordnung zu betrachten. Aufgrund des zeitlichen Rahmens dieser Arbeit war die Menge der möglichen Experimente beschränkt. Mögliche Zusammenhänge zwischen der *RP*-Überdeckung und codebasierten Überdeckungskriterien werden in Abschnitt 9.5 nochmals im Kontext der hier betrachteten Transformation untersucht. In Abschnitt 9.6 werden abschließend die wichtigsten Erkenntnisse der praktischen Erprobung des *RPC*-Ansatzes noch einmal kurz zusammengefasst.

9.1 Die *LSCToMPN*-Transformation

Grundlage der praktischen Erprobung der vorgestellten Verfahren aus den Kapiteln 7 sowie 8, bildet die sogenannte *LSCToMPN* bzw. *LSC-zu-MPN*-Transformation [Pat+10]. Bevor wir auf die Details dieser Abbildung zu sprechen kommen, betrachten wir zuerst kurz die wichtigsten Gründe, die für die Auswahl speziell dieser Transformation, als Ergänzung zur bisher verwendeten Beispieltransformation aus Abschnitt 4.4 bzw. Abschnitt A.3, sprechen:

- Die Transformation stammt nicht aus der Feder des Autors der hier vorliegenden Arbeit. Es ist also (weitestgehend[1]) ausgeschlossen, dass die Transformation – bewusst oder unbewusst – unter dem Eindruck von Zwischenergebnissen dieser Arbeit und insbesondere nicht im Hinblick auf die Testbarkeit mit dem hier vorgestellten Ansatz entwickelt wurde.

- Die Transformation ist äußerst umfangreich, nichttrivial und weist einige günstige Eigenschaften für die *RPC*-Bewertung auf (z. B. große Muster mit vielen Elementen, exogene Transformation, die dynamische Semantik sowohl der Quell- als auch der Zielsprache ist gut dokumentiert). Sie stellt einen der wesentlichen Beiträge [Pat14b] bzw. Grundlagen [Pat14a] zweier erfolgreicher Dissertationen dar, und bildet die erste Hälfte eines komplexen Codegenerierungsproblems.

- Die Transformation löst eine praktisch relevante Aufgabe, deren Umfang und Anforderungen durchaus mit „industriellen“ Problemstellungen verglichen werden kann. Sollte der *RPC*-Ansatz mit einem Beispiel dieser Größe zurecht kommen, spricht dies dafür, dass viele weitere Transformationen ebenfalls „funktionieren“ werden.

- Für die Transformation existierte bereits vor dem Start der hier durchgeführten Experimente eine initiale Testmenge (insgesamt 14 JUnit-basierte Tests) mit semantisch sinnvollen Eingabemodellen sowie passenden erwarteten Ausgaben. Die Tests weisen dabei bereits exemplarisch und auf Grundlage des Ein- und Ausgabeverhaltens (Black-Box-Tests) die Funktionalität der Transformation nach.

- *LSCToMPN* ist das Resultat eines lange laufenden Entwicklungsprojektes mit mehreren Iterationen. Die Transformation kann als weitestgehend stabil und ausgereift angesehen werden.

[1] Die *LSCToMPN*-Transformation entstand teilweise parallel zu den Ergebnissen dieser Arbeit am gleichen Institut. Die beiden Autoren der Transformation sowie der Autor dieser Arbeit waren über mehrere Jahre hinweg direkte Kollegen, allgemeine Diskussionen und ein stetiger Gedankenaustausch im täglichen Miteinander inklusive.

Um den Umfang der *LSCToMPN*-Transformation besser abschätzen zu können, sind in Tabelle 9.1 und, darauf aufbauend, in Abbildung 9.1 wichtige Kenngrößen der im Vergleich zur bisher betrachteten *BD2Ja*-Transformation sowie zu zwei umfangreichen, *Fujaba*-basierten Transformationen (die Werte stammen aus [BWW12]) gezeigt. Die Werte aus [BWW12] sind allerdings so geartet, dass Vorsicht beim direkten Vergleichen angeraten ist. Es bestehen, wie in Abschnitt 4.3 dargelegt, gewisse Unterschiede in den eingesetzten Sprachen (*SDM* gegenüber *Story-Diagramme*), was sich beispielsweise in der unterschiedlichen Art und Weise, wie Java-Fragmente verwendet werden, niederschlägt. Außerdem wurden in [BWW12] die kompletten Metamodelle berücksichtigt, und nicht nur die Teile, die unmittelbar die Transformation beschreiben; das erklärt die viel größeren Zahlen bei Paket- sowie Klassenanzahlen in Tabelle 9.1 für die *MOD2-SCM*- und *CodeGen2*-Transformationen. Letztendlich kann auch hier eine der Hauptbeobachtungen aus [BWW12] nicht wirklich bestätigt werden: die durchschnittliche Anzahl von Mustern pro Diagramm sowie die durchschnittliche Mustergröße übersteigt bei beiden hier betrachteten Transformationen deutlich die Werte aus [BWW12]. Somit weisen die hier betrachteten Transformationen andere Eigenschaften auf als die in [BWW12] untersuchten, was ein Stück weit eines der Hauptargumente für die vorgebrachte These entkräftet, dass programmierte Graphtransformationen der *Fujaba*-Interpretation oder vergleichbar, ein vergleichsweise niedriges Abstraktionsniveau – vergleichbar zu Java-Code – besitzen.

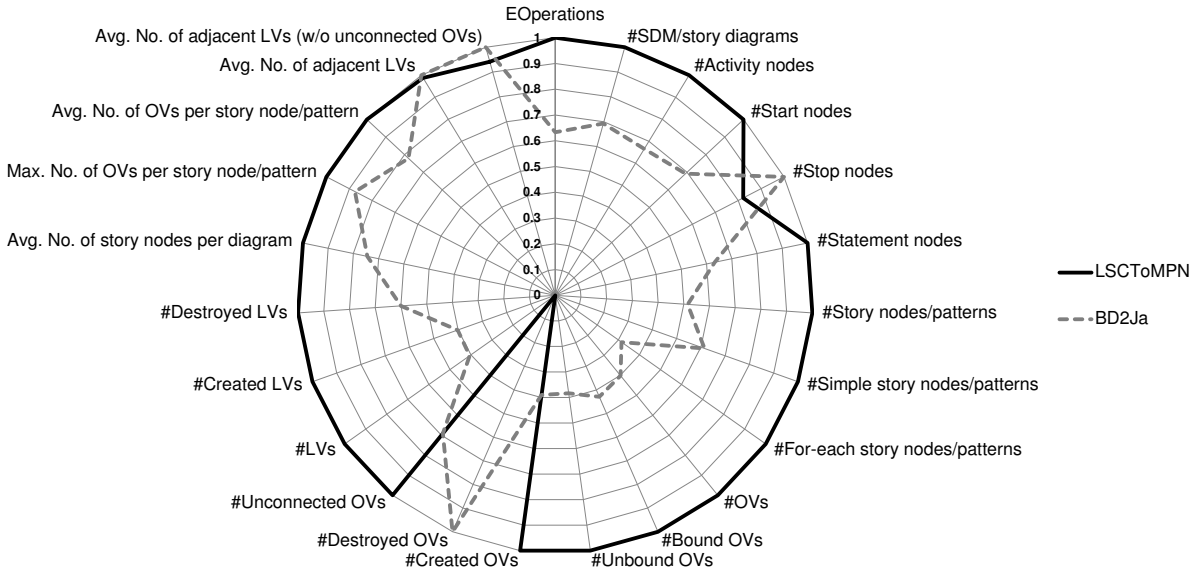
Im Netzdiagramm aus Abbildung 9.1a ist gut zu erkennen, dass *LSCToMPN* deutlich umfangreicher ist als die *BA2Ja*-Transformation – die Ausnahmen bzgl. der Anzahl der zu löschende *OVs* sowie die Anzahl der Stop-Knoten sind absolut gesehen gering und ergeben sich aus den unterschiedlichen Transformationseigenschaften (BA2Ja ist beispielsweise zum Teil endogen). Dabei weisen beide Transformationen ähnlichen Durchschnittswerten für die Anzahl an Story-Knoten pro Operation („Avg. No. of story nodes per diagram“) auf. Die jeweiligen Werte für die Anzahl an *OVs* pro Story-Knoten („Avg. No. of OVs per Story node/pattern“) sowie für die Anzahl inzidenter *LVs* pro *OV* („Avg. No. of adjacent LVs“ und „Avg. No. of adjacent LVs (w/o unconnected OVs)“) sind noch ähnlicher. Im direkten Vergleich zu den Transformationen aus [BWW12] aus Abbildung 9.1b, erkennt man, dass die beiden hier betrachteten Transformationen zueinander ähnlicher sind, als die anderen beiden, und sich von diesen auch deutlich von den hier betrachteten unterscheiden. Eine hypothetische Erklärung für die sich abzeichnenden drei Transformationsklassen wäre, dass die drei unterschiedlichen Entwicklergruppen, die hinter (i) den hier betrachteten Transformationen, (ii) *CodeGen2* sowie (iii) *MOD2-SCM* stehen, unterschiedliche Modellierungsstile nutz(t)en.

Kommen wir nun zur Darstellung der eigentlichen *LSCToMPN*-Transformation aus [Pat+10], vgl. dazu auch [Pat14b; Pat14a]. Es handelt es sich um eine Abbildung von ursprünglich *Live Sequence Charts (LSCs)* [DH01], mittlerweile von sogenannten *Extended LSCs (eLSCs)* [PPS11] – beides Varianten von *MSCs* – auf sog. *Monitor Petri Nets (MPNs)*, vgl. [Pat+10], eine spezielle Art von Petri-Netzen. *LSCToMPN* ist Teil eines umfassenden Ansatzes zur modellbasierten Entwicklung sog. *Laufzeitmonitore* für Eingebettete Systeme im Automotive-Kontext, Stichwort *AUTOSAR* [Pat+13], auf Grundlage eines *MBSecMon*[2] genannten Ansatzes. Dabei geht es, grob gesagt, darum, die Sicherheit vernetzter Eingebetteter Systeme gegenüber bewussten bzw. unbeabsichtigten Manipulationen dadurch zu erhöhen, dass die Kommunikation über den Nachrichtenkanal konti-

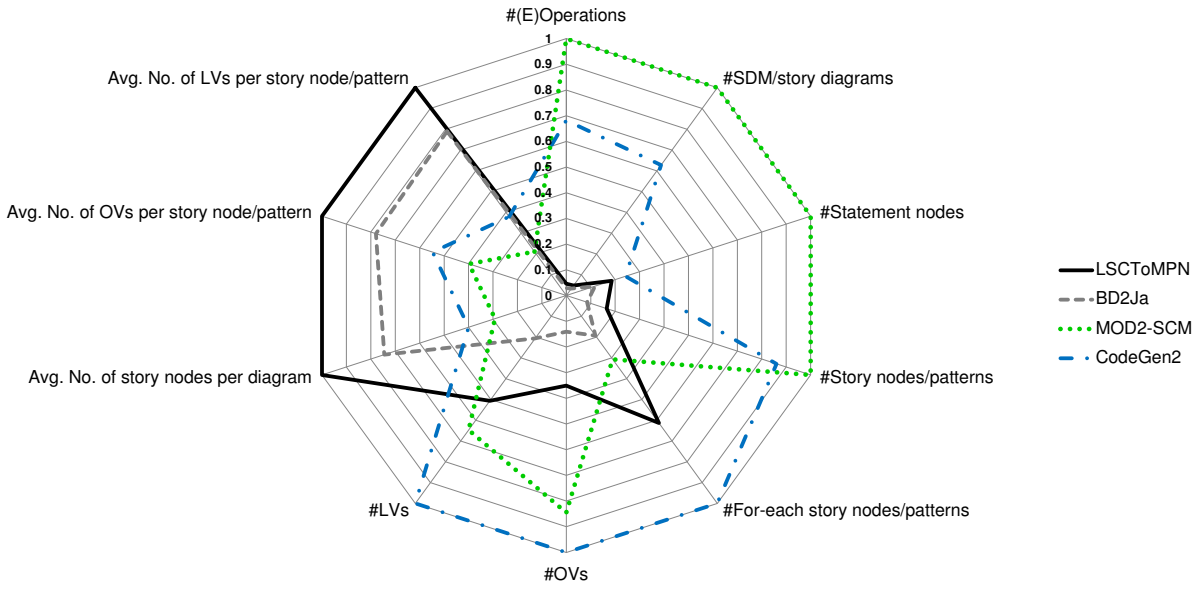
[2] Model-based Security/Safety Monitor [Pat+13]

	Transformationen			
	hier betrachtet		Vergleichswerte aus [BWW12]	
Kenngröße	*BD2Ja*	*LSCToMPN*	*MOD2-SCM*	*CodeGen2*
#(E)Packages	1	1	68*	18*
#EClassifiers	5	**10**	-	-
#(E)Classes	5	**10**	175*	162*
#(E)Operations	19	**30**	650	443
#SDM/Story diagrams	18	**26**	540	339
#Activity nodes	189	**284**	-	-
#Start nodes	18	**26**	-	-
#Stop nodes	**56**	46	-	-
#Statement nodes/patterns	31	**49**	264	64
#Story nodes/patterns	84	**163**	988	850
#Simple story nodes/patterns	67	**109**	-	-
#For-each story nodes/patterns	17	**54**	27	88
#OVs	281	**700**	1688	1997
#Bound OVs	118	**275**	-	-
#Unbound OVs	163	**425**	-	-
#Created OVs	48	**123**	-	-
#Destroyed OVs	**6**	0	-	-
#Unconnected OVs	29	**42**	-	-
#LVs	231	**568**	725	1121
#Created LVs	100	**249**	-	-
#Destroyed LVs	6	**10**	-	-
Avg. No. of story nodes per diagram	4,67	**6,27**	1,83	2,51
Max. No. of OVs per story node/pattern	14	**16**	-	-
Avg. No. of OVs per story node/pattern	3,35	**4,29**	1,71	2,35
Avg. No. of adjacent LVs	**1,64**	1,623	-	-
Avg. No. of adjacent LVs (w/o unconnected OVs)	**1,83**	1,726	-	-
Avg. No. of LVs per story node/pattern	2,75	**3,48**	0,73	1,32

Tabelle 9.1: Kenngrößen der beiden *SDM*-Transformationen im Vergleich zueinander (größere Werte jeweils **fett**) sowie Vergleichswerte für zwei Fujaba-basierten Transformationen (mit * gekennzeichnete Werte nur bedingt vergleichbar)



(a) Visualisierung der relativen Verhältnisse der Werte zwischen *BD2Ja* und *LSCToMPN* (absolute Werte wurden auf das jeweilige Maximum normiert)



(b) Alle vier Transformationen im Vergleich (nur die vergleichbaren Kenngrößen wurden berücksichtigt; auch hier erfolgt jeweils eine Normierung anhand des größten Wertes)

Abbildung 9.1: Visualisierung der Werte aus Tabelle 9.1 in Netzdiagrammform

nuierlich durch den Monitor beobachtet wird, so dass dabei auf unerwünschte Ereignisse entsprechend reagiert werden kann, vgl. auch [Pat+10]. Was zulässige und was unzulässige Nachrichten sind, entscheidet sich dabei nicht nur anhand der Art der Nachricht, sondern bestimmt sich im Wesentlichen auch anhand des Zeitpunktes des Sendens sowie dem aktuellen Systemzustand und dem zugrundeliegenden Kommunikationsprotokoll. Zur Modellierung bzw. Beschreibung entsprechender zulässiger Nachrichtensequenzen eigenen sich insbesondere *LSCs*/*eLSCs*. Direkt ausführbar auf den Zielsystemen sind diese allerdings in der Regel nicht, weshalb sie hier schrittweise in Code – beispielsweise (System)C – übersetzt werden müssen. Bei dem gewählten Zwischenformat, was als Eingabe für den eigentlichen Codegenerierungsprozess dient, handelt es sich um *MPN*-Modelle.

Die Transformation umfasst insgesamt 26 Operationen, deren Funktionalität mittels *SDM* beschrieben wurde, und die über einen relativ komplex anmutendes Abhängigkeitsgeflecht miteinander in Beziehung stehen. Im Anhang ist der Call-Graph der vollständigen Transformation in Abbildung B.1 visualisiert.

Einstiegspunkt der Transformation ist die Operation `createMPNDocumentFromMUCDocument`, die aus einer gegebenen `MUCDocument`-Instanz, also der *eLSC*-Beschreibung, eine `MPNDocument`-Instanz, das *MPN*-Artefakt, schrittweise und durch Delegation an Hilfsoperationen ableitet. Darüber hinaus wird ein Trace-Modell (in Form einer `CorrespondenceModel`-Instanz) aufgebaut, das auch zur Steuerung des Übersetzungsprozesses benötigt wird. Die Existenz eines Trace-Modells ist eine mögliche Erklärung dafür, dass die durchschnittliche Elementeanzahl pro Story-Knoten bei der *LSCToMPN*-Transformation im Vergleich zu den anderen aufgeführten Transformationen aus Tabelle 9.1 relativ hoch ist.

9.2 Praktische Anwendbarkeit der Implementierungen

Bei der Untersuchung der *praktischen Anwendbarkeit* der Implementierungen des *RPC*- und des Mutationsrahmenwerks müssen beide Fälle separat betrachtet werden. Dies ist einerseits der verschiedenartigen Umsetzungen und Techniken – klassische Implementierung der *RP*-Generierung mit Java, Instrumentierung und Auswertung der Überdeckung während der Ausführung bei der Testausführung für *RPC* gegenüber *SDM*-basierter Implementierung der Mutantenerzeugung, repetitives Ableiten von Code für jede mutierte Variante der *GT* sowie Testausführung im Stile einer Stapelverarbeitung beim Mutationsrahmenwerk – und andererseits den unterschiedlichen Anforderungen bezogen auf Qualität, Reifegrad und Benutzbarkeit der Implementierungen geschuldet.

9.2.1 RPC-Rahmenwerk

Als erste Herausforderung einer praktischen Erprobung des *RPC*-Rahmenwerks mit der in Abschnitt 9.1 vorgestellten Beispieltransformation erwies sich die Wahl des Orakels. Die existierenden Tests nutzten zu Beginn der Experimente einen Modellvergleich mittels *EMFCompare* auf der Ausgabeseite, um die tatsächlichen Ergebnisse mit vorberechneten Ergebnissen zu vergleichen. Analog zu früheren gemachten Erfahrungen, vgl. [WAS14], stellte sich das Orakel als zu empfindlich heraus, da bereits kleine Abweichungen zwischen erwartetem und tatsächlichem Ergebnis zu fehlgeschlagenen Tests führten, mit der Gefahr eines Fehlalarms.

Würde man ein solch sensitives Orakel im Rahmen einer Mutationanalyse einsetzen, bestünde die Gefahr, dass man viele der Mutanten nur zufällig „findet“, tatsächlich aber natürliche und vernachlässigbare Abweichungen überbewertet, mit der immanenten Gefahr von Falsch-Positiv-Meldungen in Form einer Identifikation von im Hinblick auf die vorhandenen Tests äquivalenten Mutanten. Eine sinnvolle Bewertungen von Testmengen ist dadurch nur eingeschränkt bis gar nicht möglich, da einerseits die Möglichkeit besteht, dass eine weitere Steigerung der tendenziell zu hohen, übersteigerten Erkennungsleistung durch weitere Tests, egal wie gut diese sind, kaum mehr möglich ist – es wurden dann bereits zu viele Mutanten (aufgrund falscher Annahmen) „erkannt“ – oder aber die Qualität der Testmenge wird bei einer beobachtbar deutlichen Zunahme der Erkennungsleistung systematisch überschätzt.

Als Konsequenz aus dieser Erkenntnis wurde ein angepasster Vergleichsalgorithmus für die resultierenden *MPN*-Modelle entwickelt, welcher der Mehrdeutigkeit, bezogen auf gültige Ausgaben, Rechnung trägt, indem er diese, in gewissen Grenzen, als äquivalent betrachtet. Unterschiede bezogen auf die Reihenfolge von Elementen in Listen der Modellrepository-Datenstruktur oder auch Unterschiede, die sich durch eine leicht veränderte Übersetzungsreihenfolge, bedingt durch unterschiedlich generierte Suchpläne, die zwischen verschiedenen Generierungsläufen auftreten können, werden so ignoriert.

Ein wichtiger Aspekt der *RPC*-Generierung besteht darin, dass sich aufgrund der Instrumentierung zur Messung der Überdeckung das wesentliche Verhalten (hinsichtlich der Ein- und Ausgabe) nicht ändern sollte. Insbesondere sollten sich keine Unterschiede ergeben, die dazu führen, dass sich die instrumentierte und die nicht-instrumentierte Variante in ihrem Ein- und Ausgabeverhalten wesentlich unterscheiden. Um zu zeigen, dass sich für das konkrete Beispiel keine Änderungen ergaben, wurde das Orakel so abgeändert, dass die erwarteten Ergebnisse der Transformation nicht mehr vorgegeben wurden. Statt dessen wurde eine uninstrumentierte Variante der Transformation, die hierfür als *hinreichend korrekt* angenommen wurde, als Referenzsystem eingesetzt. Dazu wurden dann die instrumentierte und die nicht-instrumentierte Variante mit der gleichen Eingabe stimuliert, und im Anschluss die Ausgaben verglichen. Sollte bei einer der beiden Varianten eine Ausnahme auftreten, so wird verlangt, dass in der anderen Variante die gleiche Ausnahme auftritt. Der Fall nicht-terminierender Läufe wird in der Praxis, insbesondere bei der Mutationsanalyse, durch einen vorgegebenen Time-out abgefangen. Da hier von sinnvollen, sprich in praktikabler Zeit terminierenden Transformationen ausgegangen wird, wird der Fall, dass beide Transformationsvarianten in den Time-out laufen, nicht betrachtet.

Nachdem das Orakelproblem für den speziellen Fall dieser Transformation ausreichend gut gelöst wurde, bestand der nächste Schritt in der Ableitung der *RPs* aus der ursprünglichen Transformationsbeschreibung. Der erste Schritt besteht hierbei grundsätzlich darin, die Menge der *RPs* aus der Transformation abzuleiten. Nachdem die ursprüngliche Transformationsbeschreibung leicht angepasst wurde, um problematische Stellen auszumerzen, die weiter unten beschrieben sind, zeigte die Implementierung, trotz des Umfangs der Transformation, keine Auffälligkeiten bezogen auf die Analyse der gegebenen Transformationsbeschreibung und der Ableitung der *RPs* sowie der Erzeugung der *SDM*-Implementierungen der Auswertungsoperationen. Dieser Teil des Prozesses war mit einer Laufzeit von ca. 7,4 s [3] deutlich schneller als die sich anschließende Codegenerierung so-

[3] Gemessen auf einem System mit Intel Core2-Duo P8600 CPU (Taktfrequenz 2,4 GHz, 8 GB RAM).

wie weitere Bearbeitungsschritte mit vermehrtem Festspeicherzugriff (Größenordnung: mehrere Minuten). Bezogen auf den kompletten *RP*-Generierungsschritt (inkl. Codegenerierung usw.) kristallisierten sich allerdings zwei grundsätzliche Problembereiche heraus: (i) in der Implementierung des Rahmenwerks traten einige kleinere Fehler aufgrund von bis dahin unberücksichtigten Sonderfällen zu Tage, (ii) gewisse Annahmen bzw. Richtlinien bezogen auf die Art der Modellierung in *SDM*, wie beispielsweise Namenskonventionen, wurden von der *LSCToMPN*-Transformation stellenweise verletzt. Die wesentlichen Punkte bezüglich (i) waren, dass (a) *LVs*, bei denen Quelle und Ziel ein und derselbe gebundene Knoten ist, bisher nicht berücksichtigt wurden, (b) die Menge an Parametern für die *RP*-auswertenden Operationen zu viele Einträge enthielt, da überflüssigerweise auch versucht wurde, für ungebundene *OVs* Belegungen zu übergeben, dass (c) primitive Typen (als Parameter der ursprünglichen Operation) nicht richtig an die *RP*-auswertenden Operationen übergeben wurden, was durch eine explizite Behandlung leicht korrigiert werden konnte, und dass (d) bei der Bestimmung der Parametertypen für die *RP*-auswertenden Operation bei gebundenen *OVs* der jeweilige lokale Typ der *OV* innerhalb der gerade betrachteten Regel Vorrang hatte, gegenüber anderen Verwendungen. Tatsächlich muss, aufgrund des genutzten Codegenerators für *SDMs*, der nominale Typ der Variablen analog zur „früheste Verwendung“ im Kontrollfluss bestimmt werden. Bei Punkt (ii) sind problematische Modellierungsvarianten, die deshalb als problematisch anzusehen sind, weil sie Annahmen der *RPC*-Maschinerie verletzen, von solchen zu unterscheiden, die grundsätzlich als zweifelhaft anzusehen sind, und in *SDM*-Beschreibungen vermieden werden sollten, da ansonsten die Gefahr von fehlerhaftem Verhalten besteht. Beispiele für Probleme erster Art sind Sonderzeichen im Namen sowie Namenskollisionen bei Story-Knoten. Grundsätzlich war und ist es möglich, mehreren Story-Knoten eines Diagramms den gleichen Namen zu geben. Da allerdings die *RPC*-Maschinerie aus diesen Namen Operationsnamen (und somit indirekt auch Java-Methodennamen) generiert, müssen diese in der Praxis frei von Sonderzeichen und eindeutig sein. Fehler dieser Art wurden manuell korrigiert und werden im Folgenden nicht weiter untersucht. Möglichkeiten zum Umgang mit solcherlei Einschränkungen bestünden darin, einerseits die *RPC*-Implementierung robuster zu machen oder andererseits die Einhaltung gewisser Richtlinien einzufordern. Dagegen sind Probleme zweiter Art schwerwiegender; sie deuten auf echte Probleme hin, und sind zumindest als „Bad-Smell“ [Fow99, Kap. 3] anzusehen.

Für die konkrete *LSCToMPN*-Transformation wurden zwei problematische Situationen durch die Anwendung der *RPC*-Maschinerie identifiziert:

1. Innerhalb eines *SDM*-Diagramms wurden an verschiedenen Stellen namensgleiche, ungebundene *OVs* mit unterschiedlichen Typen verwendet (in der Art `x:X` gegenüber `x:Y`). Dies ist deswegen ein Problem, da bei der Codegenerierung für alle *OVs* mit gleichem Namen genau eine Java-Variable im globalen Gültigkeitsbereich der für die Operation angelegten Methode entsteht. Diese Variable trägt den Namen (hier „`x`“) als Bezeichner, kann aber nur von einem bestimmten Typ sein (im Beispiel also entweder `X` oder `Y` bzw. deren jeweiliges Java-Pendant). Letzterer wird mehr oder weniger zufällig (in Abhängigkeit des konkreten Kontrollflussgraphen und dessen Traversierung im Rahmen der Codegenerierung) festgelegt, was im besten Fall zu statischen Fehlern beim Kompilieren mit dem Java-Compiler führt. Im schlechteren Fall besteht zwischen den gewählten Typen eine direkte Vererbungs-

beziehung und der allgemeinere Typ kommt im Code zum Einsatz. Dies führt dazu, dass das Generat vom Compiler kommentarlos akzeptiert wird, was zu undefiniertem Verhalten zur Laufzeit führen kann. Genau ein solcher Fall war im Beispiel vorhanden, und wurde durch die Einführung separater *OVs* mit unterschiedlichen Namen aufgelöst.

2. Innerhalb eines *SDM*-Diagramms wurde an einer Stellen eine ungebundene *OV* `x:X` vom Typ `X` eingeführt und an einer späteren Stelle als gebundene Variante `x:Y` vom Untertyp `Y` verwendet. Diese Art der Verwendung entspricht einer *impliziten* Typumwandlung, die potentiell auch fehlschlagen kann. Dabei kann die Auswertung des zugehörigen Musters an dieser eher unerwarteten Stelle scheitern, bei einer eigentlich schon gebundenen *OV* ohne weitere erkennbare Bedingungen. Vorzuziehen ist deshalb eine explizite Typumwandlung mittels Binding-Ausdruck, vgl. hierzu S. 67 ff.

Dem Ableiten der *RPs* inklusive des Einbindens in Auswertungsoperationen folgt die Codegenerierung für eben jene *RPs*. Hierbei ergaben sich einige praktische Herausforderungen, die zu einem Großteil mit dem eingesetzten Codegenerator in Verbindung stehen. So stellte sich heraus, dass die Anzahl der generierten *RPs* (946) und die sich daraus ergebende Anzahl an Coverage-Items ($1892 = 946 \cdot 2$) bzw. die Anzahl an Operationen zur Auswertung ($1109 = 946 + 163$) [4] dazu führten, dass die Codegenerierung zu einem langwierigem, mehrminütigem Prozess wurde. Dies ist allerdings für das eigentliche Testvorhaben in der Praxis nur dann ein Problem, wenn die Transformation sehr oft geändert werden muss – z. B. aufgrund von vielen entdeckten Fehlern oder aufgrund häufiger Änderungen. Ansonsten ist die Generierung der Coverage-Items als einmaliger, initialer Aufwand zu betrachten.

Zum anderen verstärkte sich im Rahmen der umfangreichen Codegenerierung der Verdacht, dass sich die bekannten, aber leider in der hier genutzten Werkzeugversion nicht leicht zu beseitigende Speicherlecks[5] innerhalb des Codegenerators zu einem echten praktischen Problem entwickeln könnten. (Auf die genauen Ursachen soll hier nicht eingegangen werden. Das Problem hängt, grob gesagt, mit statischen Klassenvariablen zusammen, deren Typen Java-Collections dynamischer Größe entsprechen.) Insbesondere die Visualisierungskomponente zur Darstellung generierter *RPs* war aufgrund der angedeuteten Skalierungsprobleme nur eingeschränkt nutzbar.

Was sich dagegen in der Praxis als sehr robust und skalierbar herausstellte, und somit problemfrei nutzen ließ, war die Eclipse-basierte Java-Werkzeugkette sowie das *EMF*-Rahmenwerk (inklusive dessen Codegenerators). Selbst die umfangreichste generierte Java-Klasse, die knapp unter 1900 Booleschen Variablen und jeweils entsprechend vielen Getter- und Setter-Methoden sowie reflektiven, generischen Zugriffsmethoden umfasst, und in Summe aus etwas über $117 \cdot 10^3$ *LOC* besteht, wird beispielsweise ohne Probleme im Java-Editor geladen und angezeigt sowie von der restlichen Werkzeugkette kompiliert und ausgeführt.

Zum nächsten Schritt, der Instrumentierung mit der anschließenden erneuten Codegenerierung, bleibt festzuhalten, dass dieser Teilaspekt keine Auffälligkeiten aufwies und

[4] Für jedes der 946 *RPs* eine Operation plus eine Operation pro einem der 163 Story-Knoten.

[5] In nachfolgenden Versionen wird dieses Problem adressiert und durch einen neuen Codegenerator behoben.

problemlos funktionierte. Allerdings kann aufgrund der erneuten Codegenerierung grundsätzlich nicht ausgeschlossen werden, dass sich für manche Muster ein leicht veränderter Suchplan ergibt. Auch aus diesem Grund musste das ursprüngliche Orakel entsprechen angepasst werden, um tolerant zu sein gegenüber den sich daraus eventuell ergebenden Unterschieden. Eine weitere, recht offensichtliche Konsequenz aus der zusätzlichen Instrumentierung besteht im Anwachsen des Umfangs des entsprechenden Generats. Bezogen auf den Umfang in *LOC* ergab sich für das Beispiel eine absolute Steigerung von ursprünglich 9933 hin zu 15 960 Zeilen für die „Transformator"-Klasse, die alle Operationen der Transformation umfasst, was einem relativen Zuwachs von 60,67 % entspricht. Bildet man das positive Delta zwischen den beiden *LOC*-Werten, und teilt den Wert durch die Summe aus Anzahl einfacher Story-Knoten (109) und dem Doppelten der Anzahl der For-Each-Knoten ($108 = 2 \cdot 54$), so ergibt sich ein Wert von ungefähr 28 zusätzlichen Zeilen Instrumentierungscode pro einzelner Delegation an die Auswertungsoperation, was in etwa dem Wert von 25 *LOC* entspricht, den das eingebaute hierfür genutzt Template umfasst. Die Diskrepanz ergibt sich durch das automatische Umbrechen langer Codezeilen, die sich in der Praxis im Falle einer großen Anzahl an Methodenparametern ergeben. Das mit 25 Zeilen relativ lang erscheinende Instrumentierungsfragment ergibt sich aufgrund der hierbei genutzten Indirektion zur Anbindung des aufzurufenden Codes mittels Java-Reflection (und der damit verbundenen Behandlung von Exceptions) – ursprünglich aufgrund der höheren Flexibilität und aus technischen Sachzwängen heraus gewählt. Es könnte in einer zukünftigen Versionen der *RPC*-Implementierung durch direkte (statische) Abhängigkeiten ersetzt werden.

Der letzte Schritt zur tatsächlichen Bestimmung der *RP*-Überdeckung besteht in der Ausführung der Testmenge unter Rückgriff auf den generierten Code zur *RPC*-Integration für JUnit, vgl. Abschnitt C.2. Hierbei sorgte das ursprüngliche Orakel durch Zufall dafür, dass ein Fehler in der `ReportingTestRule`-Implementierung aufgedeckt wurde: im Falle, dass jeder Test fehlschlägt, wurde keine Coverage-Information mehr an die Auswertungslogik weitergeleitet, da der Ablauf aufgrund der ausgelösten Ausnahmen vorzeitig abgebrochen wurde. Nach einer entsprechenden Korrektur, deren Ergebnis bereits am Ende des Listings C.3 zu sehen ist (`finally`-Teil in der `apply`-Methode), konnten erstmalig die Überdeckung für *LSCToMPN* bestimmt werden. Dabei stellte sich heraus, dass die Werkzeugunterstützung gemäß den Erwartungen funktionierte, und die Überdeckungsinformationen an die Eclipse-Komponente übermittelt und entsprechend visualisiert wurden. Somit konnte gezeigt werden, dass die *RPC*-Maschinerie auch für weniger synthetisch anmutende Transformationsbeispiele in der Praxis funktioniert. Auf die konkreten Messergebnisse gehen wir nach der Beschreibung der Anwendungserfahrungen bezüglich des Mutationsrahmenwerks ein.

9.2.2 Mutationsrahmenwerk

In diesem Abschnitt widmen wir uns nun der Anwendung des Mutationsrahmenwerks im Kontext des *LSCToMPN*-Beispiels. Das Augenmerk liegt dabei auf der Evaluation verschiedener Eigenschaften der Implementierung. Die Grundlage bildet die Betrachtung der grundsätzlichen Funktionstüchtigkeit und der ersten Erzeugung von Mutanten für das Beispiel. Auch hierbei gibt es verschiedene Zwischenschritte des Generierungsprozesses, die gesondert voneinander analysiert werden können. Grundsätzlich sollte neben der Funktionstüchtigkeit auch überprüft werden, wie sich das System in Hinblick auf die

Benutzbarkeit in der Praxis bewährt. Insbesondere die Dauer der Mutantenerzeugung und der manuelle Aufwand sind von Interesse.

Der erste Schritt zu einer Menge von mutierten Varianten der *SDM*-Transformation besteht in der Analyse der Transformationsimplementierung und der Suche nach Anwendungsstellen für alle unterstützten Mutationsoperatoren. Dieser Schritt ist notwendig, um dem Benutzer die Optionen für mögliche, konkrete Mutationsschritte für eine Auswahl präsentieren zu können. Dieser Schritt funktionierte für das gegeben Beispiel reproduzierbar tadellos. Die Dauer dieses Schrittes ist mit durchschnittlich 212 ms (17 Messungen, Median: 169 ms) vernachlässigbar.

Dabei werden für das Beispiel bereits knapp über 3000 potentiell mögliche Mutationen identifiziert, von denen dann tatsächlich auch 2964 ausgeführt werden können, wie weiter unten noch genauer erläutert. Die für die tatsächliche Generierung der veränderten Transformationsmodelle benötigte Laufzeit steigt, im hier betrachteten Größenbereich an durchzuführenden Mutationen, in etwa quadratisch mit der Anzahl der Mutationen an, wie in Abbildung 9.2 erkennbar. Die von der Matlab-Funktion `polyfit` berechneten Werte für die Koeffizienten a_2, a_1, a_0 des entsprechenden Ausgleichpolynoms[6] vom Grad 2 ergeben sich zu: $a_2 \approx 5{,}478 \cdot 10^{-6}$, $a_1 \approx 0{,}025$ und $a_0 \approx -0{,}328$. Somit ist der quadratische Term relativ klein. Es bleibt zu vermuten, dass über die durchgeführten Mutationsschritte hinweg einige der Datenstrukturen so anwachsen, dass sich dadurch das beobachtete Verhalten ergibt. Die Laufzeit ist aus Benutzersicht wohl noch als ausreichend gering anzusehen, so dass auch das mehrfache, wiederholte Generieren von mutierten Transformationsmodellen noch keine allzu abschreckende Tätigkeit darstellt.

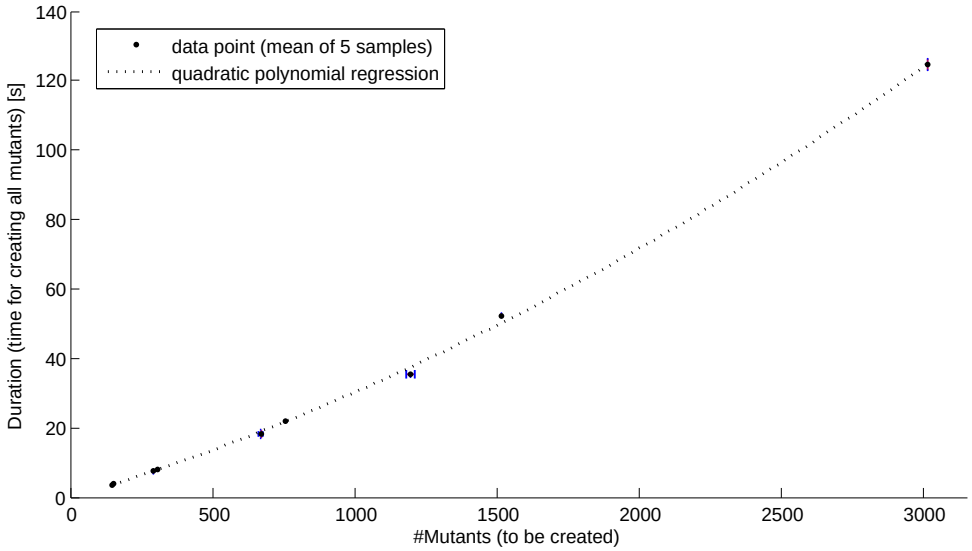


Abbildung 9.2: Dauer der Mutantengenerierung über der zu generierenden Anzahl

Der anschließende Codegenerierungsprozess, der jedes abgeleitete Transformationsmodell in eine eigene Menge an Java-Klassen übersetzt, stellt dagegen ganz andere Anforderungen an die Geduld des Anwenders. Im Vergleich zur Dauer dieses Teilschritts, sind die zuvor erwähnten Verzögerungen vernachlässigbar. Bereits bei 150 Mutanten benötigte ein

[6] $\mathrm{p(x)} = a_2 \cdot x^2 + a_1 \cdot x^1 + a_0 \cdot x^0$

Intel Core-i7-2600 basiertes System (3,4 GHz Taktfrequenz, 16 GB RAM) über 8 min für die Codegenerierung – bei allen möglichen Mutanten sind es über 6 h.[7] In Abbildung 9.3 ist der Verlauf der benötigten Zeit für die Codegenerierung über der Anzahl Transformationsvarianten (entspricht hier der Anzahl der erzeugten Mutanten) aufgetragen. Auch hierbei zeigt sich wiederum eine gute Approximation des naheliegenden Zusammenhangs durch ein quadratisches Interpolationspolynom. Die Messungen lassen den Einsatz des Mutationsrahmenwerks als Grundlage zum *mutationsbasiertem Testen* in der Praxis als unrealistisch erscheinen, da der Codegenerierungsschritt dabei wiederholt auszuführen wäre, beispielsweise nach Fehlerkorrekturen. Es bleibt offen, inwiefern ein neuer Codegenerator dieses Problem auszuräumen vermag. Im Rahmen einer Mutationsanalyse zur Validierung anderer Testansätze allerdings ist die praktische Anwendbarkeit durchaus gegeben, da der dann einmalige Aufwand zur Codegenerierung tolerabel ist.

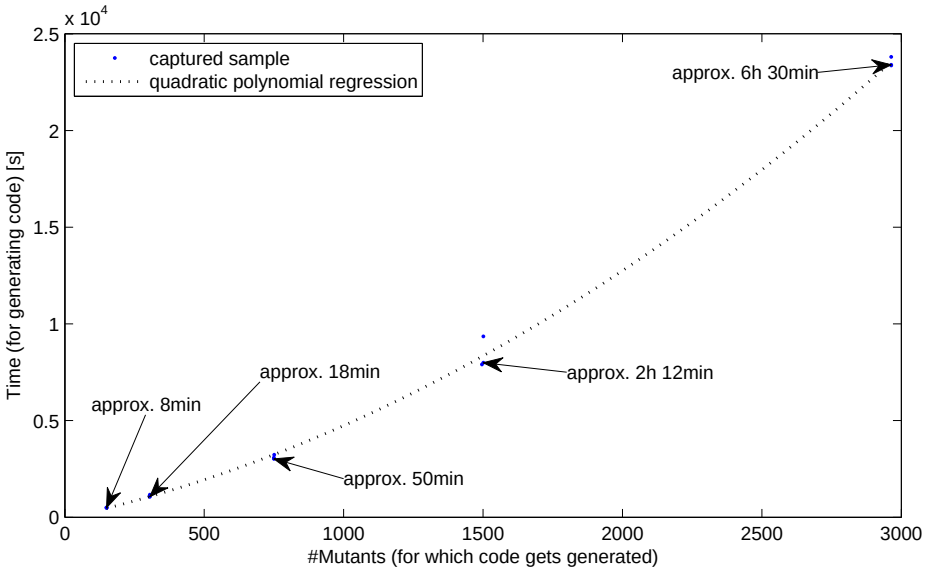


Abbildung 9.3: Dauer der Codegenerierung über der Anzahl zu erzeugender Mutations- bzw. Codevarianten

Die benötigte Gesamtlaufzeit zur Ausführung aller Tests auf jeder Variante steigt erwartungsgemäß linear mit der Anzahl der verfügbaren Varianten. Für die knapp 3000 Varianten liegen die Laufzeiten im Bereich zwischen 14 und 16 h, ebenfalls gemessen auf dem Intel Core-i7-System. Die beiden größten Anteile daran haben das repetitive Kompilieren des Java-Codes sowie die Serialisierung der Testprotokolle. Beide Teilschritte sind I/O-lastig und könnten ggf. durch den Einsatz von Halbleiter-basiertem Speicher beschleunigt werden. In der Praxis ist dies allerdings nicht nötig, da auch bei einer weniger großen Mutantenanzahl bereits sinnvolle Aussagen getroffen werden können.

[7] Eine parallel entstehender neuer Codegenerator lässt sehr viel kürzere Generierungszeiten als realistisch erscheinen.

Darüber hinaus stellten die bereits zuvor erwähnten Speicherlecks den Ansatz zur Mutation in Frage, da hier bereits nach einigen wenigen Dutzend Generierungsläufen selbst mehrere Gigabyte an zugesichertem Heap-Speicher für die ausführende Java-VM nicht mehr ausreichten, um einen kompletten Satz an mutierten Transformationen zu erzeugen und einen entsprechenden `OutOfMemoryError` zu verhindern. Bei den üblichen Einsatzszenarien des Generators stellte der Speicherverbrauch in der Vergangenheit typischerweise kein drängendes Problem dar. Im Kontext der Mutationsanalyse war das Problem nicht länger zu ignorieren. Der vermeintlich einfachste Weg, die Ursache zu identifizieren und den Generator entsprechend zu korrigieren, stellte sich als nicht-trivial heraus. Allerdings konnte im Zuge einiger Experimente und mit Hilfe eines Profiling-Werkzeugs[8] das Problem in mehrtägiger Arbeit soweit entschärft werden, dass es selbst für den Fall, dass alle möglichen Mutationen tatsächlich durchgeführt werden – also bei knapp 3000 Generierungsläufen – in der Praxis nicht mehr relevant war.

Die Ausführung der gegebenen (initialen) Testmenge auf jedem der Transformationsvarianten funktionierte in der überwiegenden Anzahl der Fälle reibungslos. Allerdings erwies sich in einigen der Fällen die Entscheidung, relativ aussagekräftige, dafür aber lange Dateinamen für die Transformationsmodelle zu generieren, teilweise als problematisch. Aufgrund von Limitierungen durch das eingesetzte Betriebssystem Windows 7 und das zugrunde liegende *NTFS*-Dateisystem, wurden zu Beginn der Experimente sporadisch einzelne Testprotokolle bei der Testausführung nicht richtig generiert. Problematisch war, dass Dateinamen auftraten, die mehr als 255 Zeichen aufwiesen. Dabei blockierte auch der zur Testausführung genutzte Ant-Task, was allerdings insofern kein gravierendes Problem darstellte, da der gesetzte Timeout dazu führte, dass die Ausführung fortgesetzt wurde und nur die akut betroffene Variante nicht weiter zu Ende getestet wurde. Letztendlich konnte das Problem durch ein Abschneiden zu langer Namen und dem Anhängen eines eindeutigen Suffixes so gelöst werden, dass alle generierten Varianten auch getestet wurden, und die jeweiligen Ergebnisse im Testprotokoll vermerkt wurden.

In der Praxis erwies sich das generierte Testprotokoll als so umfangreich und detailliert, dass es ohne Weiteres schwer zu interpretieren war. So ist beispielsweise das Protokoll einer Ausführung von 14 Tests auf jeweils allen Elementen einer Menge von 284 Mutanten eine ca. 4,2 MB große `xml`-Datei – mit einer Länge von über $5 \cdot 10^4$ Zeilen. Aus diesem Grund wurde einerseits die Ausführungslogik zur automatisierten Durchführung der Mutationsanalyse so ergänzt, dass am Ende eine Kurzzusammenfassung mit der Anzahl der erkannten und der betrachteten Mutanten erzeugt wird. Andererseits wurde ein kleines Programm erstellt, das aus der `xml`-Darstellung des Protokolls einen kondensierten Bericht erzeugt, der gegenüber der Kurzzusammenfassung auch noch Informationen darüber liefert, wie viele Mutanten durch Kompilierungsfehler respektive durch fehlgeschlagene Tests aufgefallen sind.

Abschließend bleibt festzuhalten, dass sich der im Rahmen dieser Arbeit entwickelte Mutationsansatz für *SDM*-basierte Transformationen zur Durchführung einer Mutationsanalyse als geeignet darstellt. Für ein mutationsbasiertes Testen erscheint der zeitliche Aufwand für die momentan genutzte naive Art der Codegenerierung als zu groß. Der Grund hierfür liegt in der enormen Redundanz, die diese beinhaltete, da stets der komplette Satz an Klassen und Schnittstellen neu generiert wird.

[8] `jvisualvm`, Bestandteil des Java7-*SDK*.

9.3 Testen mit dem RPC-Ansatz

Nachdem die im vorherigen Abschnitt beschriebenen technischen Hindernisse, die der Anwendung der *RPC*-Methodik auf die *LSCToMPN*-Transformation initial entgegenstanden, erfolgreich ausgeräumt werden konnten, war der nächste Schritt der Evaluation, die Methodik bei der Verbesserung der Testmenge anzuwenden und entsprechende Erfahrungen zu sammeln und zu bewerten.

9.3.1 Ausgangssituation

In Tabelle 9.2 sind in den beiden mit „initial tests“ überschriebenen Spalten die Überdeckungswerte der ursprünglichen Testmenge angegeben. Die linke Spalte „cov.“ gibt die absolute Anzahl an überdeckten Coverage-Items für jede mit *SDM* spezifizierte Operation der Transformation an, die rechte „pct.“-Spalte dagegen das Verhältnis von überdeckten zur Gesamtzahl an Coverage-Items (vgl. Spalte „total“) in Prozent. Im Anhang sind die zugrunde liegenden Messwerte in Abbildung B.2a nochmals in Form der Eclipse-*UI*-Visualisierung gezeigt.

In den Zeilen zu den markierten Operationen `createStandardPlacesForSyncSelfMessage` sowie `linkMPNExtensionPointsToBaseMPNs` ist abzulesen, dass die initiale Überdeckung 0 % beträgt, was bedeutet, dass diese beiden Operationen überhaupt nicht durch die Tests ausgeführt werden, obwohl sie grundsätzlich erreichbar wären, wie anhand von Abbildung B.1 zu erkennen ist. Somit ist ersichtlich, dass das hier verwendete Überdeckungsmaß, unabhängig von den konkreten Werten der Überdeckung, bereits in der Lage ist, ungetestete Teile der Transformation aufzudecken.

Darüber hinaus sind die beiden zusätzlich markierten Operationen `createSyncTransitionAndTranslateContentOfSyncElement` sowie `translateExtFragments` mit 37,5 % respektive 34,4 % verhältnismäßig gering abdeckt, insbesondere beispielsweise im Vergleich zu den 61,1 % Abdeckung der (ebenfalls markierten) `createMPNDocumentFromMUCDocument`-Operation, die den Einstiegspunkt der Transformation darstellt.

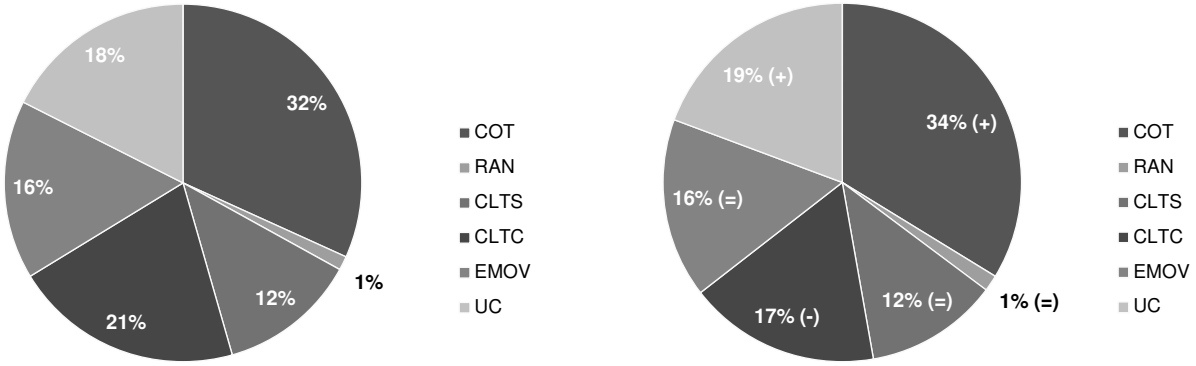
Bei echten Testvorhaben würde man sich naheliegender Weise zuerst darauf konzentrieren, die als besonders wichtig erkannte Funktionalität zu testen. Als zweite Orientierungshilfe würde man vorzugsweise die nicht oder nur unzureichend getesteten Komponenten betrachten, da in diesen die meisten unentdeckten Fehler zu vermuten sind. Für die konkrete *LSCToMPN*-Transformation liegen keine gesicherten und für das Testvorhaben nutzbaren Erkenntnisse darüber vor, was besonders wichtige und kritische Teilaspekte der Transformation sind. Zudem ist die initiale Testmenge sehr wahrscheinlich unter genau solchen Überlegungen entstanden, so dass die Hypothese, dass die kritischsten Stellen der Transformation bereits als getestet anzunehmen sind, nicht übermäßig abwegig erscheint. Somit liegt das Hauptziel des Testvorhabens darin, die als wenig getestet erkannten Bereiche durch weitere Tests zu überprüfen. Dazu wurden im Rahmen der Evaluation die in Tabelle 9.2 markierten Operationen untersucht. Im rechten Bereich der Tabelle sind ebenfalls die Werte nach dieser Optimierung gezeigt.

Bevor im nächsten Unterabschnitt der Optimierungsschritt dediziert dargestellt wird, soll zuvor noch kurz auf die Verteilung der verschiedenen Arten an *RP*-Coverage-Items für die konkrete Beispieltransformation eingegangen werden. Von den acht umgesetzten *RP*-Generierungsstrategien, vgl. Tabelle 7.1, können im Rahmen der *LSCToMPN*-Transformation sechs tatsächlich zur Ableitung benutzt werden; die beiden Strategien,

		initial tests		optimized tests		
Operation	total	cov.	pct.	cov.	pct.	Δ [pp]
appendChartPlacesToSyncTransitions	150	89	59,3 %	92	61,3 %	2,0
connectMPNExtensionPoints	50	34	68,0 %	38	76,0 %	8,0
createChartPlaces	20	12	60,0 %	12	60,0 %	0,0
createInitialPlaces	12	8	66,7 %	8	66,7 %	0,0
createMPNDocumentFromMUCDocument	90	55	61,1 %	74	82,2 %	23,7
createStandardPlacesForAsyncMessage	114	73	64,0 %	73	64,0 %	0,0
createStandardPlacesForSyncMessage	158	82	51,9 %	90	57,0 %	5,1
createStandardPlacesForSyncSelfMessage	118	0	0,0 %	60	50,8 %	50,8
createSyncTransition	204	126	61,8 %	135	66,2 %	4,4
createSyncTransitionAndTranslateContentOfSyncElement	88	33	37,5 %	55	62,5 %	25,0
createTransitionsForForbiddenChart	74	36	48,6 %	36	48,6 %	0,0
createTransitionsForIgnoreChart	74	46	62,2 %	46	62,2 %	0,0
finalizeMPN	50	31	62,0 %	31	62,0 %	0,0
handlePendingPlaces	24	14	58,3 %	15	62,5 %	4,2
syncForSyncronizableElement	98	66	67,3 %	76	77,6 %	10,3
syncObjectIdsOnTransition	28	18	64,3 %	20	71,4 %	7,1
syncPrechart	32	16	50,0 %	16	50,0 %	0,0
syncToTerminalPlaces	76	53	69,7 %	57	75,0 %	5,3
transform	48	35	72,9 %	37	77,1 %	4,2
translateElementAtLocation	54	40	74,1 %	42	77,8 %	3,7
translateExtFragments	154	53	34,4 %	71	46,1 %	11,7
translateIfThenElse	144	89	61,8 %	91	63,2 %	1,4
translateSubchart	8	4	50,0 %	6	75,0 %	25,0
translateSynchronizableElement	12	9	75,0 %	9	75,0 %	0,0
linkMPNExtensionPointsToBaseMPNs	18	0	0,0 %	14	77,8 %	77,8
helperIsSubchartOrMainchart	4	3	75,0 %	4	100,0 %	25,0

Tabelle 9.2: RP-Überdeckung der einzelnen Operationen bei *LSCToMPN* („total“: Gesamtanz. an Coverage-Items, „cov.“: überdeckte Items, „pct.“: Überdeckung in Prozent, „Δ [pp]“: Steigerung in Prozentpunkten)

die sich auf optionale Elemente beziehen, kommen aus Ermangelung solcher Konstrukte in der Transformationsbeschreibung nicht zum Einsatz. Abbildung 9.4a zeigt die anteilsmäßige Verteilung. Zu erkennen ist, dass die meisten Coverage-Items dadurch entstehen, dass der Typ einer Objektvariable aufgrund der COT-Strategie geändert wurde. Dagegen kommt die RAN-Strategie aufgrund der seltenen Verwendung von *NAC*-Elementen nur sporadisch zum Einsatz. Die restlichen Coverage-Items verteilen sich auf die übrigen Strategien, wobei die UC-Strategie einen guten Referenzwert bildet, da sie für jeden *Stroy*-Knoten genau einmal angewendet wird.



(a) Anteile der generierten *RP*-Arten bezogen auf die Anzahl insgesamt

(b) Anteilige Überdeckung nach *RP*-Arten (In Klammern: Hinweis auf Tendenzen im Vergleich zur Verteilung aus (a)[9])

Abbildung 9.4: Verteilung der Anteile *RP*-Arten bei der *LSCToMPN*-Transformation (Reihenfolge von oben nach unten bzw. im Uhrzeigersinn ab „12 Uhr“)

Interessant erscheint in diesem Zusammenhang auch die Fragestellung, ob Unterschiede existieren, bezogen auf die anteilige Abdeckung der unterschiedlichen Arten der erzeugten *RP*-Coverage-Items. Einen ersten Hinweis hierzu kann Abbildung 9.4b liefern: es ist dargestellt, wie sich die anteilige Verteilung der *überdeckten* Coverage-Items pro Strategie (jeweils bezogen auf die Gesamtmenge aller überdeckten Coverage-Items) ergibt. Der Anteil Coverage-Items, die aufgrund der COT-Strategie entstanden sind, und die durch die initiale Testmenge überdeckt wurden, beträgt 34 % bezogen auf alle überdeckten Coverage-Items (unabhängig von den Ableitungsstrategien, die ihrer jeweiligen Erzeugungen zugrunde lagen). Dies ist eine Steigerung gegenüber dem Anteil, den die COT-*RPs* bezogen auf allen generierten *RPs* einnehmen. Eine Vermutung, die sich daraus ableiten lässt, ist, dass es verhältnismäßig leichter möglich ist, COT-basierte Coverage-Items zu überdecken. Insbesondere z. B. im Vergleich mit den CLTS-basierten, deren Anteil bezogen auf die Überdeckung kleiner ist, als der entsprechende Gesamtanteil.

9.3.2 Optimierungsschritt

Das angestrebte Ziel der Testmengenoptimierung lag in der Maximierung der *RPC*-Überdeckung mit schwerpunktmäßiger Betrachtung der in Tabelle 9.2 markierten Operationen. Die zentrale Fragestellung hierbei bestand darin, zu untersuchen, wie schwer –

[9] ‘=’: unverändert, ‘+’: Zunahme, ‘−’: Abnahme

im Sinne von wie aufwendig, aber auch unter Betrachtung des Aspekts der praktischen Erfüllbarkeit – es sei, die Überdeckung möglichst umfangreicht zu steigern.

Insgesamt wurden in vier kurz hintereinander liegenden Phasen versucht, die Überdeckung zu steigern. Dabei wurde unterschiedlich vorgegangen und mehrere Taktiken zugrunde gelegt:

1. **Ad-hoc-Vorgehen** – Zu Beginn war das Wissen zur und das Verständnis der Funktionsweise der Transformation noch rudimentär. Der erste Schritt lag somit darin, anhand der Metamodell- und Transformationsbeschreibung sowie des Call-Graphen (vgl. Abbildung B.1) und im Zusammenspiel mit den bereits existierenden Tests syntaktisch korrekte Modelle als Eingabe abzuleiten, und so zu versuchen, die Überdeckung zu steigern. Dabei wurden die vorhandenen Tests nicht mitausgeführt, um die Überdeckung nur anhand der selbst erstellten Tests zu bestimmen. Hierbei stellte sich heraus, dass die Rückmeldung durch die gesammelten Überdeckungskennzahlen indirekt dabei helfen kann, die Funktionsweise und die Ausführung der Operation besser nachzuvollziehen. Andererseits war bereits nach zwei bis drei Tests eine Sättigung der erreichten Überdeckung zu beobachten, bei der weitere, vermeintlich andersartige Tests keine Verbesserung der Überdeckung mehr brachten. Der Einsatz der *RPC*-Überdeckung erfordert also – so die ersten Erfahrungen – in der Praxis ein zielgerichteteres, methodischeres Vorgehen.

2. **Testen nach Analyse der jeweiligen *RPs*** – Der nächste Schritt lag darin, bestehende Testmodelle so abzuändern, dass an einer bestimmten Stelle des Transformationsprozesses ein noch nicht überdecktes Coverage-Item neu überdeckt wurde. Dazu musste analysiert werden, welche Eingabe in der Lage ist, die Transformation soweit zur Ausführung zu bringen, dass dabei eine bestimmte *SDM*-Regel erreicht wird. Mit Hilfe des Debuggers ließen sich entsprechende Eingaben, falls vorhanden, leicht identifizieren. Allerdings war eine zielgerichtete Modifikation aufgrund unterschiedlicher Sachverhalte schwierig. Zum einen müssen die benötigten Anpassungen an der Eingabe meist über eine Sequenz von Verarbeitungsschritten stabil und erhalten bleiben (ohne das sonstige Verhalten negativ zu beeinflussen). Darüber hinaus führen isolierte Änderungen am Eingabemodell häufig zu inkonsistenten Modellen, was die Transformationsausführung unter Umständen unmöglich macht. Und letztendlich muss möglichst sichergestellt werden, dass die durchgeführten Änderungen an der entsprechenden Stelle auch tatsächlich das gewünschte Verhalten stimulieren. Insgesamt muss die Funktionsweise der Transformation detailliert analysiert werden. Insbesondere diese Tatsache ist allerdings als positiver Nebeneffekt des Testvorhabens zu sehen, da sich so ggf. konzeptionelle Schwächen der Implementierung zeigen können. Im konkreten Fall konnte so innerhalb einer Operation ein gravierender Fehler dergestalt entdeckt werden, dass die Operation keinerlei schreibende (Seiten-)Effekte umfasste, dafür aber rekursive Aufrufe beinhaltet, was in diesem Zusammenhang potentiell zu nichtterminierendem Verhalten führen kann. Im Rahmen dieser Phase traten die meisten Schwierigkeiten offen zutage. Ohne detailliertere Kenntnisse der Transformation konnte für einzelne Transformationen die Überdeckung zwar isoliert gesteigert werden, komplexere Zusammenhänge blieben so allerdings unberücksichtigt.

3. **Testen mit semantisch sinnvollen Tests** – In der vorherigen Phase zeichnete sich als Hauptproblem ab, dass es praktisch schwierig ist, die Transformation mit semantisch sinnvollen Eingabemodellen so anzuregen, dass bestimmte Zustände im Ablauf erreicht werden. Aus diesem Grund wurde nun versucht, zusammen mit den Autoren der *LSCToMPN*-Transformation, weitere Tests zu erzeugen. Ein wesentlicher technischer Unterschied zur vorherigen Situation ergab sich durch den Einsatz eines mächtigen Editors für die *LSC*-Sprache, mit dessen Hilfe konsistente aber auch entsprechend umfangreichere und komplexere Modell erzeugt werden konnten. Die Detailkenntnisse der Transformationsprogrammierer war darüber hinaus wertvoll, um semantische Feinheiten der Abbildung zu erfassen. So konnten bestimmte Zustände im Programm mit Hilfe der Detailkenntnisse viel leichter erreicht werden, als durch alleiniges Reengineering (auf Basis der Transformationsbeschreibung und vorhandener Testfälle). Auch war das planvolle Ändern von Testmodellen so leichter möglich, da u. a. die Überdeckung besser interpretiert werden konnte.

4. **Kombination und Bündelung** – Die letzte Phase bestand darin, die durch die Zusammenarbeit mit den *LSCToMPN*-Autoren gewonnenen Erkenntnisse soweit wie möglich eigenständig zu nutzen, um die Überdeckung weiter auszubauen. Hierbei bestätigte sich die Vermutung, dass sich in der Praxis einige der Coverage-Items grundsätzlich nicht überdecken lassen, da dies voraussetzen würde, dass man eine Modellinstanz im Verlauf der Transformation dynamisch abändern könnte – vergleichbar beispielsweise zur Manipulation von Java-Variablen während der Ausführung mit Hilfe des Debuggers.

In Tabelle 9.2 ist in den beiden mit „optimized tests“ überschriebenen Spalten das Endergebnis der Optimierung nach allen Phasen dargestellt. In der Spalte am rechten Rand ist darüber hinaus das Delta (Δ) in Prozentpunkten zwischen den initialen prozentualen Werten und den prozentualen Werten angegeben. Es zeigt sich, dass für die betrachteten Operationen die Überdeckung jeweils deutlich gesteigert werden konnte.

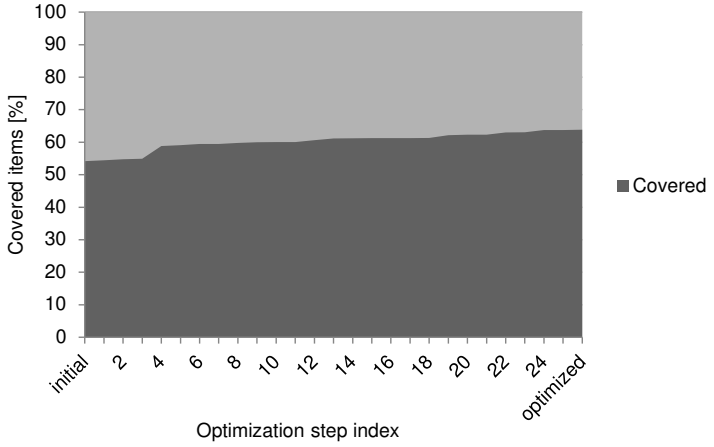


Abbildung 9.5: *RP*-Überdeckung im zeitlichen Verlauf der Optimierung

Der schrittweise Verlauf der Optimierung ist in Abbildung 9.5 in aggregierter Weise dargestellt. Jeder Schritt auf der x-Achse repräsentiert einen neu hinzugefügten Testfall;

die Testmenge wuchs dabei von initial 14 Tests auf insgesamt 40 Testfälle an. Die jeweils von der Testmenge erreichte *RP*-Überdeckung stieg von anfänglich 1025/1892 $\widehat{=}$ 54,18 % auf 1208/1892 $\widehat{=}$ 63,85 % an.

9.3.3 Erfahrungen und Erkenntnisse

Im Folgenden werden die wichtigsten, aus der experimentellen Erprobung der *RP*-Überdeckung gewonnenen Erkenntnisse kurz zusammengefasst.

Nichterfüllbarkeit von Coverage-Items

Die wesentlichste Erkenntnis aus der Anwendung ist, dass sich durchaus einige Überdeckungsanforderungen ergeben, die nicht erfüllbar sind. Zieht man den Vergleich zu klassischen Überdeckungskriterien für Quellcode, so stellt man fest, dass diese Problematik selbstredend auch dort existiert; man denke nur an Pfadüberdeckung bei Programmen mit vielen (verschachtelten) Schleifen oder aber auch an Datenflussüberdeckung, vgl. z. B. [FW88]. So können nur solche Pfade durch den Kontrollflussgraphen möglich sein, die dazu führen, dass bestimmte Eigenschaften im Modell an einem Punkt der Transformation unerfüllbar bleiben, vgl. hierzu auch den in [GWZ94] beschriebenen Zusammenhang zwischen *Feasible Path Analysis* und dem Testen. Realisiert eine Operation beispielsweise einen Teil der Funktionalität einer übergeordneten Operation, so sind die möglichen Zustände, in denen erstere Operation aufgerufen wird, abhängig vom Aufrufkontext innerhalb der zweiten Operation.

Auch stellten sich bestimmte Muster als problematisch heraus, da sie zu nicht erfüllbaren Anforderungen führten. Exemplarisch ist beispielsweise der Fall, dass die *LHS* einer Regel eine einzelne gebundene *OV* umfasst. Diese Situation tritt beispielsweise auf, wenn ein bereits gebundener Knoten mit einer neu zu erstellenden Struktur verbunden werden soll. Die Strategie sorgt dann dafür, dass eine Kopie der *LHS* als *RP* verwendet wird, bei der ebenfalls nur die gebundene *OV* „gesucht“ wird. Die Negation, also die Anforderung, dass für dieses *RP kein* Treffer existiert, kann häufig praktisch gar nicht fehlschlagen, von Ausnahmen, wie einem `null`-Parameter, Selbstkanten oder Attributbedingungen abgesehen.

Wirklich signifikante Aussagen über Anteile oder Typen der Anforderungen, die vermehrt nicht erfüllbar sind, können hier nicht getroffen werden. Die Datenlage erscheint dafür zu begrenzt, insbesondere da aus zeitlichen Gründen nicht alle Operationen hinsichtlich der Erfüllbarkeit der generierten *RPs* untersucht werden konnten. Die aus Abbildung 9.4 abgeleitete Vermutung, dass sich bestimmte Anforderungen (objektiv) leichter abdecken lassen als andere, bleibt davon unberührt bestehen.

Aus den gemachten Erfahrungen lassen sich weitere Eindrücke kondensieren:

- Durch den Kontrollfluss entstehen starke Abhängigkeiten zwischen den Mustern, teils mit negativem Einfluss auf die Erfüllbarkeit einzelner Coverage-Items in ihrem jeweiligen Kontext. Bei einer isolierter Betrachtung einzelner Regel, aus denen sie abgeleitet wurden, wäre die Erfüllbarkeit dagegen teilweise gegeben.

- Eine tiefe Verschachtelung von Operationsaufrufen erschwert das Testen zusätzlich. Mit wachsendem Abstand zur Wurzel des Call-Graphen steigt der Aufwand tendenziell stark an.

- Das isolierte Testen einzelner Operationen kann das Überdecken von *RPs* helfen zu vereinfachen; die hierzu benötigten Testmodelle müssen nicht in dem Sinne realistisch sein, als dass sie im Kontext der ganzen Transformation tatsächlich an der Stelle auch auftreten können.

- Composite-Kanten sind problematisch, da einerseits eine intakte Containment-Hierarchie vorausgesetzt wird, um Assoziationen im Rahmen eines Suchplans entgegen ihrer eigentlichen Navigierbarkeit zu traversieren; Fehler führen hierbei zu unvorhergesehenem Verhalten. Andererseits können zurzeit noch *RPs* entstehen, die Zyklen in der Enthaltenseinsbeziehung fordern.

- Detailwissen über die Funktionalität der Transformation erscheint unerlässlich für sinnvolles Testen.

Klassifikation von Coverage-Items

Die Hypothese, dass bestimmte *RPs* grundsätzlich leichter zu erfüllen seien als andere, legt nahe, dass in Bezugnahme auf die Überdeckung bzw. Nichtüberdeckung eine unterschiedlich starke Bewertung zugrunde gelegt werden sollte. Leichter zu erfüllende Anforderungen sollten beispielsweise, alleine aus ökonomischen Überlegungen heraus, vorrangig erfüllt werden. Dementsprechend müsste die bisher einheitlich betrachtete Menge an Coverage-Items in verschiedene Klassen partitioniert werden, abhängig beispielsweise vom zu erwartenden Aufwand oder auch der Fehlererkennungsleistung, deren dedizierte Untersuchung hierfür benötigt werden würde.

Umgang mit unerfüllbaren Anforderungen

Wie bereits dargelegt, sind im Allgemeinen nicht alle generierten Coverage-Items erfüllbar und eine automatisierte Identifikation unerfüllbarer Anforderungen ist auch nicht realisierbar. Dabei stellt sich die Frage, wie mit solchen Anforderungen in der Praxis umgegangen werden sollte. Eine Option – allerdings keine besonders gute – besteht darin, diese Coverage-Items einfach zu ignorieren und sich nach einer gewissen Zeit anderen Anforderungen zuzuwenden. Dies hat allerdings den Nachteil, dass entsprechende Entscheidungen ohne Weiteres nicht systematisch erfolgen und für Außenstehende nicht nachvollziehbar sind. Es ist nicht ausgeschlossen, dass sich beispielsweise andere Testautoren wiederholt dem selben Problem widmen, da die zuvor erfolglos verlaufenen Bemühungen unbekannt bleiben.

Daraus folgt, dass entsprechende Anforderungen, die sich als praktisch oder theoretisch unerfüllbar herauskristallisiert haben, zwar aus der Betrachtung aussortiert werden sollten, diese Entscheidung aber gut dokumentiert und begründet werden muss. Ein Ausschluss von der Überdeckungssteigerung bedeutet aber nicht, dass die entsprechenden *RPs* auch von der programmatischen Auswertung bei der dynamischen Ausführung ausgeschlossen werden sollten. Im Gegenteil, kann doch ein Verbleib in der Menge der ausgewerteten *RPs* dazu beitragen, dass ggf. fälschlich aussortierte Anforderungen erkannt werden können – dann nämlich, wenn sie unerwartet und zufällig doch noch erfüllt werden. Von der Berechnung der Überdeckungsmetrikwerte sollten die entsprechenden Elemente allerdings ausgeklammert werden, um die erreichbaren Werte möglichst dicht an die definitionsgemäß bestmögliche Überdeckung von 100 % bringen zu können. Zurzeit

bietet das Rahmenwerk leider noch keine diesbezügliche Unterstützung. Eine entsprechende Erweiterung böte sich im Rahmen zukünftiger Verbesserungen an.

Möglichkeiten zur Automatisierung?

Grundsätzlich wären aus Anwendersicht die Erforschung und Umsetzung weiterer Möglichkeiten zur Automatisierung wünschenswert. Insbesondere das in Abschnitt 7.4.1 angedeutete Ausfiltern von nicht zu erfüllenden *RPs*, vgl. Abbildung 7.14, Punkt ③, wäre von großem praktischen Interesse, da dies den Analyseaufwand potentiell deutlich reduzieren könnte, vgl. dazu auch [DO91]. Gegenüber dem hier genutzten proaktiven Verfahren, dass einige *RPs* gar nicht erst generiert – allerdings nur auf Grundlage von auf Regelebene lokal begrenzter Informationen – hätte ein filterbasierter Ansatz den Vorteil, dass leicht auch globale Informationen benutzt werden können, beispielsweise auch Pfadbedingungen, wie sie sich aufgrund des Pfades durch den Kontrollflussgraphen ergeben.

Ein verwandtes Problem besteht in der automatisierten Generierung von (rudimentären) Testmodellen, die sicherstellen, dass ein bestimmter Punkt in der Transformation tatsächlich erreicht wird. Lösungen hierfür hätten ebenfalls das Potenzial, den manuellen Aufwand deutlich zu reduzieren. Entsprechende Ansätzen zur Generierung von Tests für klassische Programme, beispielsweise auf Grundlage von Constraint-Solving und symbolischer Ausführung [DO91; GWZ94], existieren. Mit den in Kapitel 6.3 erwähnten Model-Finder-Werkzeugen ließen sich, ein Satz entsprechender Bedingungen vorausgesetzt, entsprechende *MM*-Instanzen ableiten. Für eine prototypische Umsetzung eines solchen Ansatzes für die *SDM*-Sprache fehlt es zurzeit im Wesentlichen an einer Komponente zur Bestimmung der benötigten Bedingungen aus der Transformationsbeschreibung, z. B. mittels symbolischer Ausführung der Graphersetzungsschritte.

Editoren und Modellierungsrichtlinien versus Erfüllbarkeit

Da bei der hier verwendeten Ableitung der *RPs* nur die transformationsspezifischen Informationen aus den jeweiligen *GT*-Regeln und dem oder den Metamodell(en) verwendet werden, können zusätzliche Nebenbedingungen, wie sie beispielsweise aus semantischen Überlegungen heraus entstehen, nicht berücksichtigt werden. Das hat den Effekt, dass durch die Überdeckungskriterien zum Teil Eigenschaften bei den Eingabemodellen gefordert werden, die zu Inkonsistenzen bezogen auf diese Nebenbedingungen führen. Positiv formuliert, werden dadurch also auch negative Tests eingefordert.

Werden allerdings die Testmodelle mit Hilfe mächtiger Editoren spezifiziert, können die in ihnen implementierten Konsistenzregeln verhindern, dass Modelle frei genug spezifiziert werden können, um alle Anforderungen abzudecken, obwohl dies ohne den Editoreinsatz möglich wäre. Genauso können Modellierungsrichtlinien, deren Einhaltung eine Vorbedingung für die zu testende Transformation darstellt, die Menge der möglichen Eingaben soweit einschränken, dass bestimmte Konstellationen praktisch nicht mehr möglich sind, obwohl die beteiligten Ein- und Ausgabesprachen diese zulassen würden.

Es bleibt festzuhalten, dass – in Abhängigkeit von der konkreten Transformationsaufgabe – eine Berücksichtigung zusätzlicher Nebenbedingungen bei der Generierung der *RPs* wünschenswert ist. Auch wäre dies eine sinnvolle Erweiterung der bestehenden Implementierung. Ein praktikabler Ansatz zur technischen Umsetzung könnte auch hier der Rückgriff auf Model-Finder-Werkzeuge sein.

Bewertung der Nützlichkeit

Basierend auf den gesammelten Erfahrungen, im Rahmen des Versuchs die Überdeckung für die *LSCToMPN*-Transformation zu erhöhen, soll kurz der subjektive Eindruck der Nützlichkeit des *RPC*-Ansatzes geschildert werden, bevor wir uns den messbaren Ergebnissen der Mutationsanalyse zuwenden.

Zum einen erscheint es durchaus realistisch, für eine Transformation des Umfangs von *LSCToMPN* die Überdeckung aller generierter *RPs* zu untersuchen. Sortiert man (praktisch und theoretisch) unerfüllbare Anforderungen aus, erscheint eine Überdeckung der verbleibenden Anforderungen in vertretbarer Zeit zu 100 % erreichbar. Unklar ist allerdings, inwiefern Änderungen an der Transformation – z. B. aufgrund von Fehlerkorrekturen – während dieses Prozesses dazu führen können, dass sich die Menge der generierten Änderungen so stark ändert, dass die erreichte Gesamtüberdeckung nicht mehr monoton ansteigt.

Darüber hinaus wurde die vorgestellte *RPC*-Überdeckungsmetrik von den *LSCToMPN*-Autoren als nützlich bewertet, da mit ihr ein Mittel zur Verfügung stand, um fehlende Testfälle zu identifizieren und die Testbemühungen systematisch zu steuern. Im Rahmen der Experimente konnten darüber hinaus einige problematische Modellierungsvarianten, zwei komplett ungetestete Operationen sowie – im Rahmen einer kurzen Sitzung – ein tatsächlich übersehener Fehler in der Transformation identifiziert werden. Ohne Zweifel, ein nützlicher Aspekt (für dieses konkrete Beispiel).

Insgesamt traten ähnliche Herausforderungen auf, wie sie auch beim Einsatz struktureller Überdeckungskriterien bei klassischen Programmiersprachen zu erwarten sind, beispielsweise (i) hohe Anforderungen bzgl. der Kenntnis der Implementierung, (ii) die Tendenz zum „Testen im Kleinen“ erschwert die Bewertung von Testresultaten, und (iii) zum Teil unerfüllbare Anforderungen. Die Werkzeugunterstützung war insgesamt ausreichend praktikabel und stabil. Ein gesteigerter Automatisierungsgrad zur Reduktion des manuellen Aufwands wäre an einigen Stellen aus Anwendersicht sicherlich wünschenswert, ist hier allerdings nicht mehr Inhalt der Arbeit. Wobei auch die Umsetzbarkeit von Testfallgenerierung und Ausfilterung nicht erfüllbarer Überdeckungsanforderungen (sehr wahrscheinlich) durch grundsätzliche, beweisbare Einschränkungen limitiert wird.

9.4 Leistungsabschätzung und -bewertung von RPC

Nachdem im letzten Abschnitt exemplarisch gezeigt werden konnte, dass ein Testvorhaben von einem Einsatz der *RPC*-Metrik profitieren kann, und dass eine Steigerung der Überdeckung in der konkreten Anwendung möglich ist, muss nun noch auf die Fragestellungen VII (Leistungsbewertung des Testverfahrens) und X (Bewertung des Gesamtnutzens) aus Kapitel 1 eingegangen werden. Hierzu wurden verschiedene Messreihen auf Basis der *LSCToMPN*-Transformation und der durch des Mutationsrahmenwerk ableitbaren Varianten durchgeführt.

In Tabelle 9.3 ist das Ergebnis der Mutationsanalyse mit allen ableitbaren Mutanten zusammengefasst. Die obere Zeile (*initial*) umfasst die Werte für die initial gegebene Testmenge. Von den 2964 Programmvarianten, mit jeweils einem bewusst eingebauten Fehler, wurden 1685 Varianten als fehlerhaft erkannt, was zu dem „rohen“ (engl. raw) Mutationswert von 56,85 % erkannten Mutanten führt. Von den insgesamt getöteten Mu-

tanten wurden nunmehr 702 durch die *Java Runtime Environment (JRE)* erkannt (aufgrund nicht zu kompilierender, fehlender Klassen). Diese relativ große Anzahl verwundert etwas, lässt sich aber auf die einfach gehaltenen Heuristiken zur Vermeidung invalider Mutanten bei deren Erzeugung zurückführen. Hier existieren offenkundig Möglichkeiten zur Optimierung durch verbesserte, aber auch aufwendigere Analysen. Bereinigt man die Menge der Mutanten um die invaliden Exemplare mit unvollständig kompilierendem Code (auf den bei der Ausführung tatsächlich zugegriffen wird) und bezieht die Anzahl der durch Tests erkannten Mutanten auf den Umfang dieser bereinigten Menge, so erhält man den normalisierten Mutationswert der äußersten rechten Spalte der Tabelle. Die untere Zeile (*optimized*) umfasst die Werte für die mittels *RPC*-Einsatz optimierte Testmenge. Hierbei sind mehrere Aspekte bemerkenswert:

1. Der Mutation-Score der hinsichtlich *RPC* optimierten Testmenge übersteigt den der ursprünglichen Testmenge signifikant.
2. Die Anzahl der aufgrund von nichtkompilierendem Code erkannten Mutanten steigt durch vermehrtes Testen ebenfalls leicht an. Dies lässt sich dadurch erklären, dass erst durch einen tatsächlichen Laufzeitzugriff auf die dann fehlenden Klassen – dies wird aufgrund der verbesserte Testmenge wahrscheinlicher, da mehr Code überdeckt wird – die vorhandenen Fehler tatsächlich zum Tragen kommen.
3. Die Steigerung der Erkennungsleistung erfolgt überwiegend aufgrund von Verbesserungen bzgl. der Testmenge, was sich in einem verhältnismäßig größeren Anteil der durch Tests entdeckten Mutanten zeigt.

		#killed_mutants			mutation score	
testset	#mutants	by Java-RTE	by tests	total	raw	normalized
initial	2964	702	983	1685	56,85 %	43,46 %
optimized	2964	737	1112	1849	62,38 %	49,93 %

Tabelle 9.3: Ergebnisse der Mutationsanalyse mit allen Mutanten

Um weiterhin ausschließen zu können, dass die Verbesserung des Mutation-Score-Wertes hier nur zufälliger Natur war, wurden weitere Messungen durchgeführt. Dazu wurden zufällige Untermengen unterschiedlichen Umfangs[10] der Gesamtmengen aller Mutanten gebildet, und jeweils der Mutationswert vor und nach der Testmengenoptimierung für die Untermengen bestimmt. Bei der Bildung der Untermengen wurde darauf geachtet, dass sich pro Operation in etwa der gleiche Anteil an ausgewählten zu möglichen Mutanten einstellte, um so einen Selektionsbias zu vermeiden.

In Abbildung 9.6 ist das Ergebnis des Vergleichs der Mutation-Score-Werte vor und nach der Optimierung über alle Messungen in Boxplot-Form dargestellt. Dabei sind wiederum die Situationen mit normierten und die nichtbereinigte (also mit vorhandenen nicht vollständig kompilierenden Mutanten) Mutantenmengen unterschieden. Es zeigt sich, dass Mutationsanalysen mit Untermengen der Menge aller möglichen Mutanten brauchbare Abschätzungen für den Mutationswert der Gesamtmenge aller Mutanten liefern (angedeutet durch die relativ kleinen Intervallgrößen in Abbildung 9.6). In beiden

[10] Es wurden Untermengen mit 5 %, 10 %, 25 % und 50 % aller möglichen Mutanten verwendet.

Situationen (*raw* und *normalized*) ist eine signifikante Verbesserung der Mutationserkennungsleistung zu beobachten, was darauf hindeutet, dass durch die Verwendung der *RP*-Überdeckung tatsächlich adäquate Tests entstanden sind. Allerdings bleibt zu betonen, dass es sich hierbei nur um *ein* konkretes (repräsentatives) Transformationsbeispiel handelt. Weitere Experimente mit anderen Transformationen sollten diese Beobachtung im Idealfall bestätigen können.

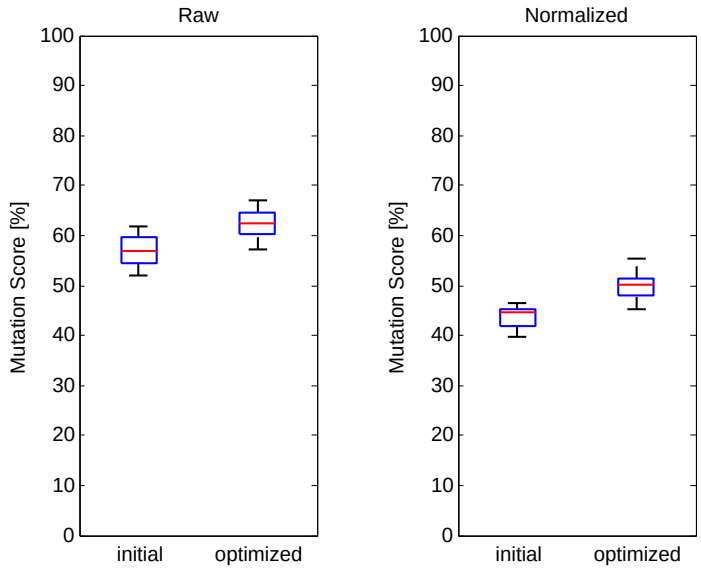


Abbildung 9.6: Mutation-Scores vor (initial) und nach (optimized) der Testmengenoptimierung (auf Grundlage der Steigerung der *RP*-Überdeckung) – links alle generierten Mutanten, rechts nur valide Mutanten (n=8)

Ein weiterer Aspekt, der ebenfalls durch Messungen untersucht wurde, ist die vermutete *Korrelation* zwischen *RPC* und dem Mutation-Score. Dazu wurden, in Anlehnung an das Verfahren aus [FW93], zehn reduzierte Testmengen jeweils durch zufällige Auswahl von zehn Tests der optimierten Testmenge, bestehend aus 40 Tests, gebildet. Für jede dieser kleinen Testmengen (im Folgenden als „small“ bezeichnet) wurde der *RPC*- und der Mutationswert bestimmt. Anschließend wurde jede kleine Testmenge durch das zufällig Hinzufügen 20 weiterer Tests (aus der Menge der übriegen Tests der optimierten Testsuite) zu einer dann 30 Tests umfassenden großen Testmenge (im Folgenden als „big“ bezeichnet) erweitert. Für diese vergrößerten Testmengen wurde dann wiederum *RPC*- und Mutationswerte bestimmt. Grundlage zur Bestimmung der Mutationswerte war eine zufällig ausgewählte Menge an Mutanten, mit einem Umfang von 5 % aller möglichen Mutanten (in absoluten Zahlen: 150 Stk.).

Es zeigte sich, dass sich in allen Fällen aufgrund des Hinzufügens weiterer Tests, eine Verbesserung der *RPC*- und Mutationswerte ergab. Interessanterweise sind die dabei auftretenden absoluten Steigerungen bezüglich der *RPC*- und Mutationswerte größenmäßig ähnlich. In Abbildung 9.7 sind im linken Bereich die Messwerte als Punktewolke dargestellt. Im Falle der kleinen Testmengen streuen die Werte stärker, da unter Umständen aufgrund der geringen Testanzahl nur sehr selektiv getestet wird. Werden zufällig gute und verschiedenartige Tests ausgewählt, kann die erreichte Überdeckung dagegen sehr

viel größer sein. Deutlich zu erkennen ist, dass die gemessenen Werte in dem hier auftretenden Überdeckungs- und Wertebereich stark miteinander korreliert sind. Ein hoher *RPC*-Wert geht mit einem großen Mutationswert einher; umgekehrt treten kleinere Werte ebenfalls gemeinsam auf. Dies ist ein weiterer Hinweis darauf, dass die auf Grundlage der *RP*-Überdeckung erstellten Testmengen wahrscheinlich als adäquat im Hinblick auf die Erkennung von Fehlern zu betrachten sind.

Im rechten Teil der Abbildung 9.7 sind die sich ergebenden absoluten Steigerungen der *RPC*- und Mutation-Score-Werte aufgrund der Vergrößerung der Testmengen dargestellt. Die Messpunkte streuen im Wesentlichen rund um den Median bzw. den Mittelwert der Messungen, was bedeutet, dass sich die beobachtbaren Steigerungen bei allen Messungen in Größe und Richtung ähneln. Durch das „Sampeln“ der Tests aus der Gesamtmenge der Tests ergeben sich folglich Teilmengen mit ähnlichen Eigenschaften, ohne dass extreme Ausreißer auftreten. Auch zeigt sich, dass zwischen den beiden Delta-Arten eine Korrelation besteht.

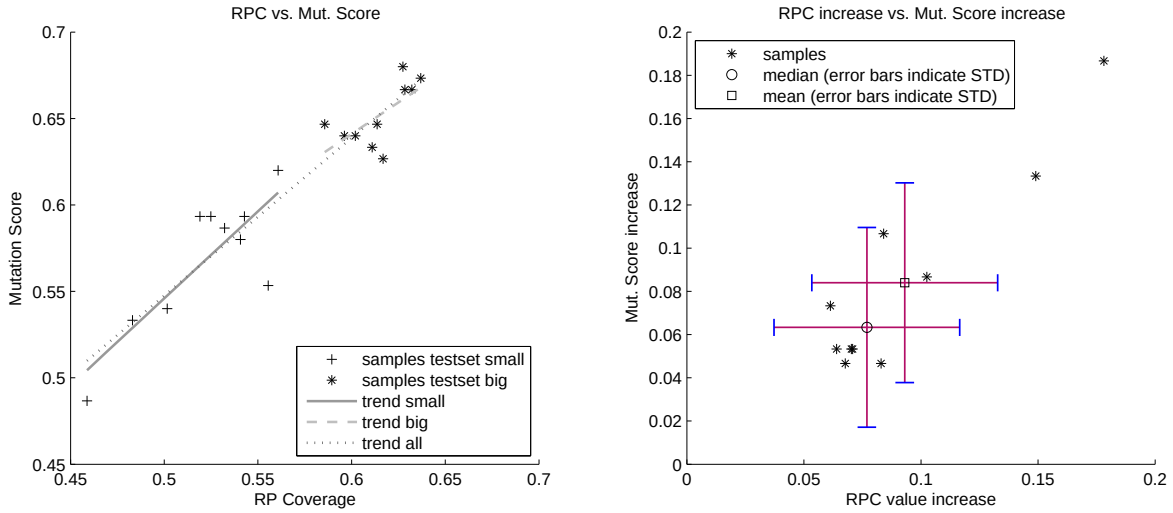
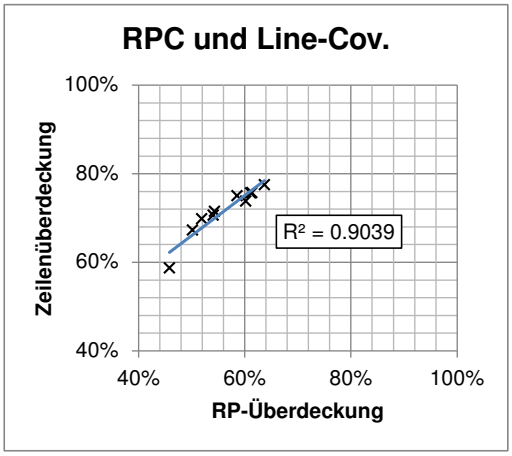


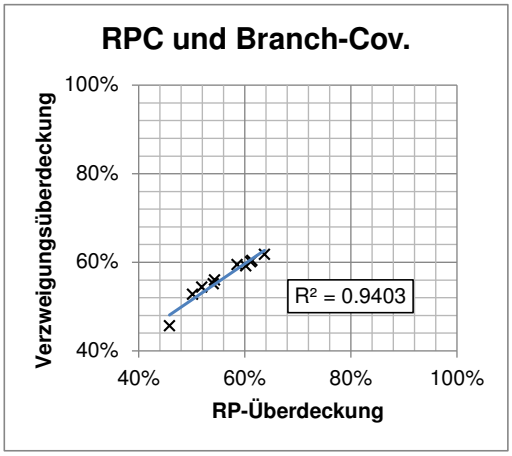
Abbildung 9.7: Zum Vergleich von *RPC* ggü. *Mutation-Score* bzw. Vergleich der jeweiligen Zuwächse

9.5 Erweiterter Vergleich: RPC vs. Codeüberdeckung

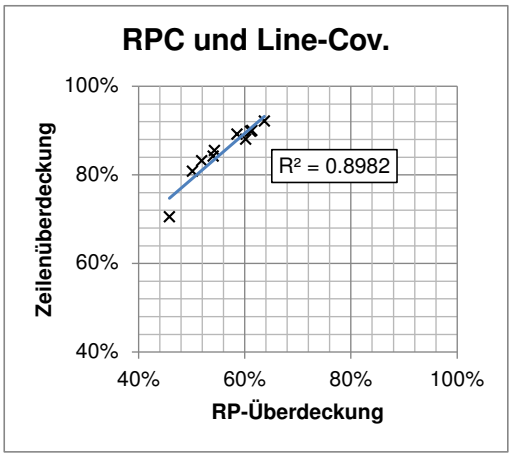
In Ergänzung des initialen Vergleichs zwischen *RPC* und codebasierten Überdeckungskriterien auf S. 188 ff., Abschnitt 7.4.2, betrachten wir hier die entsprechenden Verhältnisse im Kontext des *LSCToMPN*-Beispiels. Die Grundlagen hierfür bilden zehn Messreihen, zusammengesetzt aus je fünf für kleinere Testmengen (mit jeweils 10 zufällig ausgewählten Tests aus der Menge der optimierten *LSCToMPN*-Tests) sowie für größere Testmengen (mit jeweils 30 zufällig ausgewählten Tests). Für jede Testmenge wurde, neben der *RP*-Überdeckung, jeweils die Zeilen- sowie die Verzweigungsüberdeckung mit dem zuvor erwähnten Cobertura[11] bestimmt.

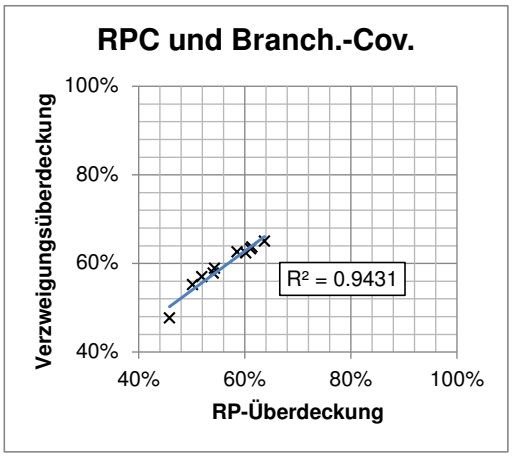
[11] `http://cobertura.github.io/cobertura/`





(a) Streudiagramme für Zeilen- bzw. Verzweigungsüberdeckung im Vergleich zu *RPC* für den Fall „Code inkl. *RPC*-Instrumentierung“





(b) Streudiagramme für Zeilen- bzw. Verzweigungsüberdeckung im Vergleich zu *RPC* für den Fall „Code ohne *RPC*-Instrumentierung“

Abbildung 9.8: Aggregierte Überdeckungswerte (über allen relevanten Operationen bzw. Methoden der `Transformator`-Klasse) für 10 zufällige Testmengen (je fünf kleinere mit 10 und fünf größere mit 30 Tests) bei *LSCToMPN*

Den Beginn macht eine grobe Analyse der grundsätzlichen Verhältnisse. In Abbildung 9.8 sind dazu vier Streudiagramme gegeben, welche die *aggregierten* Überdeckungswerte für die *RP*-Überdeckung in Kombination mit den Werten der beiden codebasierten Überdeckungsarten Zeilen- (linkes Diagramm) bzw. Verzweigungsüberdeckung (rechtes Diagramm) visualisieren. Ein einzelner Datenpunkt steht hierbei für die Überdeckung bei einer der zehn Testmengen. Der Überdeckungswert beziehen sich dabei auf die `Transformator`-Klasse. D. h. alle existierenden sowie die überdeckten Coverage-Items werden für alle relevanten Methoden bzw. Operationen gemeinsam betrachtet.

Vier (anstatt von zwei) Diagramme ergeben sich deshalb, weil in der Darstellung zwei Situationen unterschieden werden: im oberen Bereich, Teilabbildung 9.8a, sind die Werte gezeigt, die sich bei vorhandener *RPC*-Instrumentierung im Quellcode ergeben. Im unteren Bereich, Teilabbildung 9.8b, sind dagegen die codebasierten Überdeckungen ohne die *RPC*-Instrumentierung gezeigt. Es ist unmittelbar zu erkennen, dass die Codefragmente für die Instrumentierung die Überdeckungswerte der codebasierten Ansätze verringern, wobei der Effekt bei der Zeilenüberdeckung stärker ausgeprägt ist, was auf die spezielle Struktur der zusätzlichen Codefragmente zurückzuführen ist.

Bei der hier verfolgten aggregierten Erfassung über alle Methoden bzw. Operationen hinweg zeigt sich, dass in allen vier Situationen die codebasierten Überdeckungskriterien stark mit *RPC* korreliert sind. Zu beachten ist dabei allerdings, dass die Überdeckungswerte, selbst für den günstigsten Fall mit der höchsten codebasierten Überdeckung, nämlich Zeilenüberdeckung bei fehlender *RPC*-Instrumentierung, noch einen deutlich sichtbaren Abstand vom Idealwert von 100 % aufweisen. Auch erscheint das Potenzial zur Verbesserung der Überdeckung für den Fall der Zeilenüberdeckung im Vergleich zur *RP*-Überdeckung nahezu ausgeschöpft.

Kurios mutet es an, dass das absolute Niveau der Überdeckung bei der Verzweigungsüberdeckung und der *RP*-Überdeckung nahezu identisch ist, obwohl es sich um grundsätzlich verschiedenartige Überdeckungsansätze handelt. Es sei auch darauf hingewiesen, dass die zugrundeliegende Basistestmenge, aus denen die hier benutzten Teilmengen durch zufällige Auswahl abgeleitet wurden, mit dem Ziel der Steigerung der RP-Werte optimiert wurde. Eine naheliegende Hypothese aufgrund der vorliegenden Werte wäre also, dass eine Steigerung der *RP*-Überdeckung sehr wahrscheinlich auch mit einer Steigerung bei den codebasierten Kriterien einher geht.

Betrachtet man allerdings die nichtaggregierten Rohdaten der Messungen in einem Streudiagramm – Abbildung 9.9 umfasst zwei entsprechende Diagramme, bei denen jeder Messpunkt die nach der Ausführung einer einzelnen Testmenge erreichte Überdeckung pro individueller Methode bzw. Operation zeigt – so ergibt sich ein differenzierteres Bild. Für die beiden codebasierten Überdeckungsarten scheinen sich nämlich in den Bereichen höherer Codeüberdeckung praktische Grenzen für deren weitere Steigerungen herauszukristallisieren. So stagniert die Zeilenüberdeckung nährungsweise bei einen Sättigungswert von ca. 95 %. In einem gewissen Bereich ist die Zeilenüberdeckung damit nicht mehr näherungsweise proportional zur *RP*-Überdeckung, vgl. den hervorgehobenen Bereich im linken Diagramm von Abbildung 9.9. Bezüglich der Verzweigungsüberdeckung lässt sich eine ähnliche Sättigung im Bereich von 70 bis 75 % Überdeckung erahnen, vgl. den im rechten Diagramm der Abbildung hervorgehobenen Bereich. Das *RP*-Kriterium erscheint im Bereich der beiden hervorgehobenen Regionen in der Lage zu sein, die Tests differenzierter bewerten zu können, als dies durch codebasierte Überdeckungskriterien möglich ist. Was hier nicht zu erkennen ist, sich aber anhand der Rohdaten für den

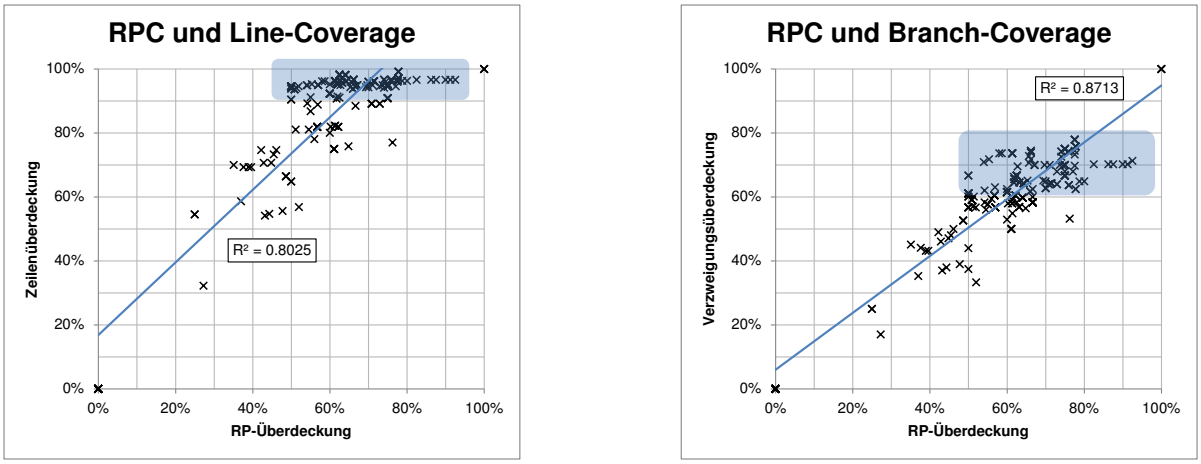
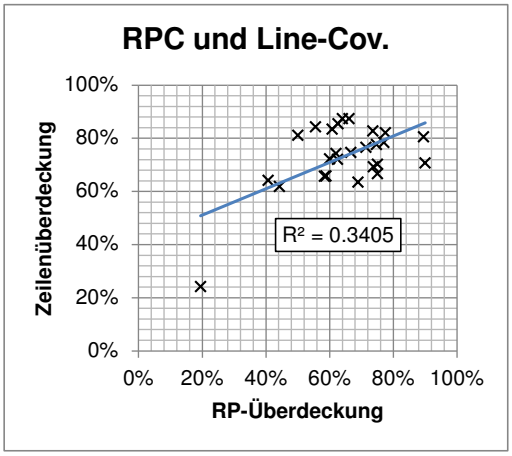


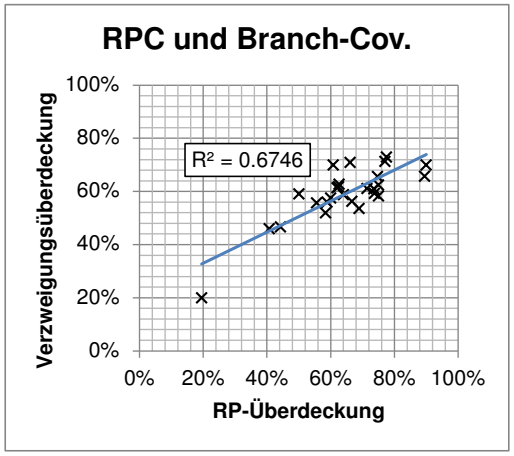
Abbildung 9.9: Streudiagramme der vollständigen Rohdaten (für den Fall „Code ohne *RPC*-Instrumentierung“) zur Verdeutlichung des Sättigungsverhaltens der Codeüberdeckung

Fall der vorhandenen *RPC*-Instrumentierung zeigte: es kann praktisch vorkommen, dass vollständige *RP*-Überdeckung für eine Operation erreicht wird, ohne dass die jeweilige Codeüberdeckungsvariante ihr Optimum erreicht. Eine Subsumption beider Codeüberdeckungsvarianten jeweils durch *RPC* ist also ausgeschlossen.

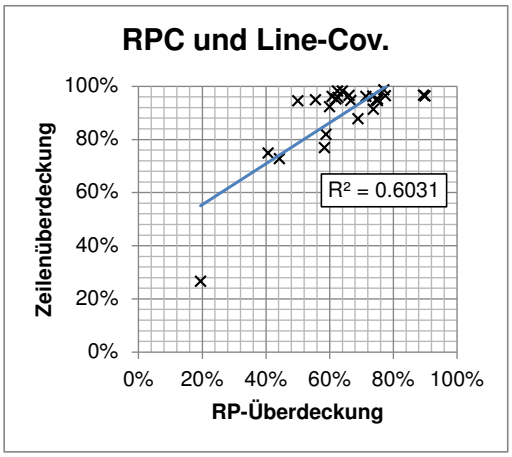
Eine weitere Vermutung lässt sich anhand der beiden Diagramme aus Abbildung 9.9 entwickeln, nämlich dass sich bei hoher Codeüberdeckung der Wert des Korrelationskoeffizienten eher reduziert. Dies wäre grundsätzlich konsistent zu den bereits beschriebenen Beobachtungen für die BD2Ja-Beispieltransformation aus Abschnitt 7.4.2. Dort erschienen die Werte der Zeilenüberdeckung und der *RPC* quasi unkorreliert und die Werte der Verzweigungsüberdeckung und der *RPC* nur geringfügig korreliert zu sein.

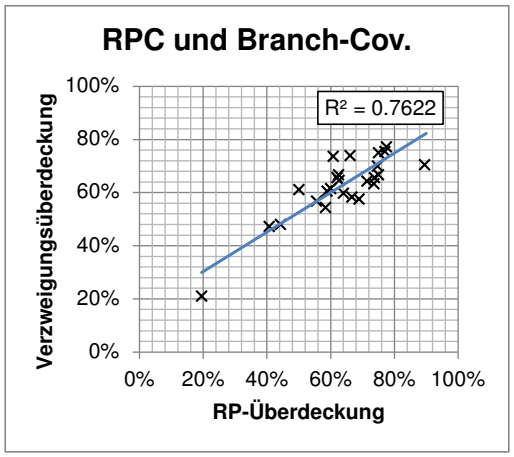
Dieser Trend zur Abschwächung einer möglichen Korrelation lässt sich tatsächlich beobachten, wenn man sich nur auf die Messwerte der tendenziell höheren Testabdeckung der fünf größeren Testmengen beschränkt. In Abbildung 9.10 sind dazu die verschiedenen Überdeckungswerte der insgesamt 26 Methoden der `Transformator`-Klasse für beide möglichen Situationen (mit und ohne Instrumentierungscode) dargestellt. Die Art der Präsentation entspricht damit genau jener aus Abbildung 7.16. Allerdings wurden hier die Überdeckungswerte der fünf Läufe (jeweils einer pro großer Testmengen) gemittelt; eine Beschränkung auf nur einen Lauf erschien zu willkürlich. Die sich ergebenden Korrelationskoeffizienten (entsprechend der jeweiligen Wurzel aus dem Bestimmtheitsmaß) sind höher als noch im Fall der BD2Ja-Transformation, was sich ggf. dadurch erklären lässt, dass sich im BD2Ja-Beispiel die absoluten Überdeckungswerte pro Methode über alle Methoden hinweg in vergleichsweise hohen Wertebereichen bewegt. Dadurch kommt die angesprochene Sättigung potentiell stärker zum Tragen. Dennoch ist die Korrelation im Zusammenhang mit hohen Codeüberdeckungsmaßen auch für das *LSCToMPN*-Beispiel in allen betrachteten Fällen geringer als in den vorherigen Diagrammen. Es bleibt zu vermuten, dass sich, aufgrund der Tendenz des sich verringernden Korrelationskoeffizienten für große Codeüberdeckungswerte, ein (linearer) Zusammenhang zwischen *RPC* und den codebasierten Kriterien für höhere Überdeckungswerte zunehmend auflöst. Für die Zeilenüberdeckung ist dies bereits deutlich zu beobachten. Für die Verzweigungsüberdeckung scheint ein potentieller Zusammenhang stabiler zu sein.





(a) Streudiagramme für Zeilen- bzw. Verzweigungsüberdeckung im Vergleich zu *RPC* für den Fall „Code inkl. *RPC*-Instrumentierung“





(b) Streudiagramme für Zeilen- bzw. Verzweigungsüberdeckung im Vergleich zu *RPC* für den Fall „Code ohne *RPC*-Instrumentierung“

Abbildung 9.10: Durchschnittliche Überdeckung für jede der 26 relevanten Methoden der `Transformator`-Klasse bei *LSCToMPN* (Messwerte aller fünf größeren Testmengen wurden dazu gemittelt)

Wie bereits dargelegt, wurden beide Transformationen vor allem mit Testmengen getestet, die unter *RPC*-Gesichtspunkten erstellt bzw. optimiert wurden. Es zeigte sich im Rahmen der Evaluation, dass sich bereits bei moderat hoher *RP*-Abdeckung eine vergleichsweise hohe Zeilen- und wertemäßig ähnliche Anweisungsüberdckung einstellen kann. Außerdem scheint die *RP*-Überdeckung, in ihrer hier eingeführten Form (und in den hier erreichten Überdeckungsbereichen auf den Beispielen), relativ stark und deutlich mit der Verzweigungs- bzw. Zweigüberdeckung auf Codeebene zu korrelieren. Somit kann das *RPC*-Kriterium aus testtheoretischer Sicht zumindest als brauchbar gelten, da es stärkere Anforderungen führt als die gemeinhin als zu schwach eingeschätzte Anweisungsüberdeckung. Interessant wäre nun ggf. noch die Untersuchung des umgekehrten Weges: wie schlagen sich Testmengen im Hinblick auf die *RP*-Überdeckung, die im Hinblick auf ihre Codeüberdeckung optimiert wurden. Aus Zeitgründen konnte eine entsprechende Untersuchung nicht mehr durchgeführt werden.

Davon abgesehen, bietet die *RP*-Überdeckung auch unbestreitbare Vorteile gegenüber den etablierten und ggf. weniger aufwendigeren codebasierten Kriterien, denn sie

- funktioniert auch im Falle eines interpretierten Ansatzes,
- ist unabhängig von der Art der Codegenerierung,
- berücksichtigt Eigenheiten des Metamodells (wie z. B. die Vererbungsbeziehungen),
- kann typische Spezifikationsfehler aufgreifen und
- produziert Überdeckungsanforderungen, die in visualisierter Musterform direkt interpretierbar sind.

9.6 Zusammenfassung

Im Rahmen der Evaluation wurden das entwickelte *RP*-Überdeckungskonzept sowie das Mutationsrahmenwerk anhand der umfangreichen und nichttrivialen *LSCToMPN*-Transformation experimentell erprobt und wichtige Eigenschaften untersucht. Das unabhängig entwickelte *LSCToMPN*-Beispiel wurde ausgewählt, um eine möglichst objektive Bewertung sicherzustellen.

Im Rahmen der Erprobung konnte die praktische Anwendbarkeit der Implementierung nachgewiesen werden. Sowohl das *RPC*-Rahmenwerk als auch das Mutationsrahmenwerk konnten die ihnen jeweils zugedachten Aufgaben zufriedenstellend erfüllen (vereinzelte Korrekturen kleinerer Fehler inbegriffen). Bei der erstmaligen Durchführung der *RP*-Instrumentierung zeigten sich einzelne problematische Stellen in der ursprünglichen Transformationsspezifikation, auf welche die *RPC*-Maschinerie empfindlich genug reagierte, so dass diese aufgedeckt und korrigiert werden konnten. Dies kann als Indiz für die Notwendigkeit guter statischer Analysen interpretiert werden.

Aus der Anwendung des Mutationsrahmenwerks lässt sich schließen, dass tendenziell sehr viele Mutanten erzeugt werden können. Dabei ist die Dauer der Suche nach potentiellen Anwendungsstellen für die spezifischen Mutationsoperatoren vernachlässigbar, insbesondere im Vergleich zur eigentlichen Erzeugung der Mutanten. Letztere war für den genutzten Aufbau so hoch, dass das mutationsbasierte Testen, speziell was die Erstellung einer mutationsadäquaten Testmenge anbelangt, für eine neu zu entwickelnde Transformation zurzeit als ungeeignet angesehen werden muss. Für die Bewertung einer bestehenden Testmenge ist die Laufzeit dagegen durchaus tolerabel. Ein neuer Codegenerator für eMoflon hätte allerdings genug Potenzial, um diese Problematik zu entschärfen.

Im Hinblick auf die Evaluation der vorgestellten Methodik zur Verbesserung der Testabdeckung bleibt festzuhalten, dass mittels *RPC* gänzlich ungetestete Transformationsteile sehr leicht identifizierbar werden. Auch deuten niedrige Werte der Abdeckung – im Falle des konkreten Beispiels Werte um die 35 % – auf nicht ausreichend getestete Aspekte hin. Das eigentliche Steigern der Überdeckung erfordert ein planmäßiges, gerichtetes Vorgehen und setzt ein tieferes Verständnis der generellen Funktionsweise *und* der Implementierung der *GT* voraus. Das Testen auf Basis von *RPC* kann somit indirekt das Verständnis einer bis dato unbekannten Implementierung fördern. Durch die intensive Auseinandersetzung mit dem *SUT* können bereits Fehler offenkundig werden, wie im konkreten Fall zu beobachten. Allerdings erweist es sich teilweise auch als nichttrivial, Testfälle so zu erstellen, dass die Auswertung einer bestimmten *GT*-Regel tatsächlich auf eine der eingeforderten Modellstrukturen trifft. Das Haupthindernis besteht darin, dass sich Sequenzen vorangehender Transformationsschritte nicht überspringen lassen. Darüber hinaus sind in der Praxis nicht erfüllbare Anforderungen durch einzelne *RPs* möglich, wobei die durchgeführte Evaluation noch nicht genügende Daten für belastbare Hypothesen über die Art und den Umfang entsprechender Anforderungen liefert. Ein anderer Aspekt der zu beobachten ist: nicht alle *RP*-Arten lassen sich gleich gut überdecken; bestimmte Anforderungen erscheinen schwieriger zu erfüllen. Eine genauere Untersuchung dieser Thematik sowie eine Untersuchung der Fehlererkennungsleistung einzelner *RP*-Klassen sind potentiell relevante Aufgaben für weitergehende Untersuchungen. Zusammengefasst lässt sich festhalten, dass die Steigerung der *RP*-Überdeckung ein grundsätzlich mögliches, zum Teil aber auch recht mühsames Unterfangen darstellt, insbesondere dann, wenn man die zugrunde liegende Transformationsimplementierung noch nicht hinreichend gut kennt. Das isolierte Testen einzelner Operationen (nicht einzelner *GT*-Regeln) kann das gezielte Abdecken ggf. erleichtern. Eventuell würde der konsequente Einsatz von Testtechniken wie *RPC* auch dazu führen, dass Entwickler mit der Zeit Implementierungen bevorzugen, welche leichter zu testen sind.

Die Durchführung der Mutationanalyse führte zu mehreren Erkenntnissen. Das wichtigste Ergebnis ist, dass eine Optimierung der Testmenge hinsichtlich der *RP*-Überdeckung zu einer beobachtbar höheren Mutationsabdeckung führt. Somit erscheinen die eingeforderten Testfälle die Mutationsadäquatheit tatsächlich zu steigern, was für die Sinnhaftigkeit des *RPC*-Ansatzes spricht. Die beobachtbare Steigerung erscheint dafür auch signifikant genug. Testreihen mit unterschiedlichen Teiltestmengen deuten zusätzlich darauf hin, dass die *RP*-Überdeckung und die Mutationsüberdeckung korreliert sind. Dabei bewegen sich die zu beobachtenden absoluten Steigerungen beider Überdeckungsarten auf, absolut gesehen, ähnlichem Niveau. Dies kann als Hinweis darauf interpretiert werden, dass *RPC* eine ausreichend gute und aus Anwendersicht sicherlich praktikablere Alternative zum Mutationstesten darstellt.

Im Rahmen einer erweiterten Untersuchung der Zusammenhänge zwischen Code- und *RP*-Überdeckung konnten für das *LSCToMPN*-Beispiel stärkere Zusammenhänge zwischen den Messwerten beobachtet werden, als für das Bd2Ja-Beispiel. Es zeigte sich, dass höhere *RP*-Überdeckungswerte tendenziell auch mit größeren Werten für Anweisungs- bzw. Verzweigungsüberdeckung einhergehen. Allerdings ist mit ansteigendem Überdeckungsniveau bzgl. der codebasierten Kriterien ein Sättigungseffekt zu beobachten, der besonders ausgeprägt bei der Zeilenüberdeckung in Erscheinung tritt, da sich diese im Zusammenspiel mit den umfangreicheren Testmengen ihrem theoretischen Optimum annähert. Im Hinblick auf eine solche Interpretation, erscheinen die in Abschnitt 9.5 vorge-

stellten Messergebnisse diese Vermutung ebenfalls zu stützen. Als weiterer Punkt konnte eine Subsumption von Anweisungs- oder Verzweigungsüberdeckung durch die *RPC* praktisch ausgeschlossen werden. Ein weiteres Ergebnis der Untersuchung liegt in der weiter zu untersuchenden Vermutung, dass *RPC* zu strikteren Testanforderungen führt als die codebasierte Anweisungsüberdeckung. Auch scheint die *RP*-Überdeckung mindestens so leistungsfähig zu sein, wie die Verzweigungsüberdeckung. Unabhängig davon bietet die *RP*-Überdeckung unbestreitbare Vorteile gegenüber den Codeüberdeckungsvarianten, wie die Unabhängigkeit vom Generat und damit die wichtige Kompatibilität mit interpretierten Ansätzen, die Berücksichtigung von Metamodelleigenheiten sowie die grafisch visualisierbaren Coverage-Items.

Zur Verallgemeinerbarkeit der hier beschriebenen Resultate sei angemerkt, dass die Experimente exemplarischen Charakter besitzen. Allerdings ist die Evaluation mit zwei Transformationen und diversen Messreihen relativ umfangreich. Auch sind die Ergebnisse in sich konsistent. Obwohl alle Experimente mit äußerster Sorgfalt und Gewissenhaftigkeit durchgeführt wurden, sind weitere unabhängige Untersuchungen empfehlenswert, um die Ergebnisse zu überprüfen. Denn statistisch auftretende Abweichungen (z. B. durch Übertragungsfehler bei der Eingabe von Messwerten) aber auch unbekannte systematische Fehler sind nicht mit absoluter Sicherheit auszuschließen. Fehlern der ersten Art wurden hier durch mehrfache Kontrollen und multiple Messungen mit Mittelungen begegnet. Die Gefahr von systematischen Fehlerquellen kann dagegen nur durch möglichst unabhängige Experimente effektiv ausgeschlossen werden.

10 Fazit

Die Motivation zur vorliegenden Arbeit lag in der Untersuchung von angepassten Testverfahren für Modelltransformationen im Allgemeinen und programmierten Graphtransformationen im Speziellen. Das Hauptziel bestand in der Entwicklung eines strukturellen Überdeckungskriteriums für letztere. Entsprechende Verfahren und Kriterien sind wichtige und entscheidende Bausteine in einem praktikablen Konzept zur Sicherung der Qualität entsprechender Entwicklungsartefakte. Der vermehrte Einsatz von *MDSD*-Techniken auch in kritischen Anwendungen verlangt zunehmend nach entsprechenden Verfahren.

Typischerweise sind testende Ansätze nur als exemplarische, beispielgetriebene Form der Verifikation zu sehen, bieten dafür aber eine vergleichsweise niedrige Eintrittsschwelle und geringe Kosten für eine potentiell große Steigerung des Vertrauens in die Korrektheit einer implementierten Funktionalität. Dabei steht und fällt die Nützlichkeit einer auf dem Testen basierenden Analyse mit dem eingesetzten Überdeckungsbegriff. Dieser steuert mehr oder weniger unmittelbar die Auswahl und die Ableitung von Tests, hilft bei der Identifizierung nur ungenügend getesteter Bereiche einer Umsetzung und limitiert ganz praktisch den ansonsten leicht ins Uferlose anwachsenden Testaufwand.

Konkret wurde in dieser Arbeit ein strukturelles Überdeckungskriterium für eine bestimmte Klasse graphtransformationsbasierter Programme vorgestellt und detailliert beschrieben, eine konkrete Umsetzung im Eclipse-Umfeld skizziert sowie die Einsatzfähigkeit, Tauglichkeit und Leistungsfähigkeit exemplarisch anhand konkreter Beispieltransformationen untersucht und gezeigt. Im Zuge dessen wurde auch der aktuelle Stand der Forschung und Technik eingehend analysiert und überblicksartig dargestellt. Hierbei zeigte sich, dass existierende Testansätze für Modelltransformatioen überwiegend als Vertreter des funktionsbasierten (Black-Box-)Paradigmas anzusehen sind und ansonsten vorwiegend auf die populären und verbreiteten Sprachen wie *Atlas Transformation Language (ATL)* oder *QVT* abzielen. Die Eigenschaften letzterer Sprachen sind allerdings nur grob mit denen von Graphtransformationen vergleichbar, weshalb angepasste Verfahren als sinnvolle Ergänzung zum Stand der Forschung anzusehen sind.

10.1 Ergebnisse und Zielerreichung

Richtet man den Blick auf die in der Einleitung formulierten Leitfragen, vgl. Fragestellung I bis X, die den groben Rahmen dieser Arbeit definieren und deren Untersuchung und Beantwortung Gegenstand der Betrachtung war, so stellt man fest, dass alle Fragestellungen bearbeitet und zum Großteil auch aussagekräftige Antworten ermittelt werden konnten. Aufgeschlüsselt anhand der einzelnen Punkte ergibt sich nun zum Ende hin das folgende Bild:

Zu Fragestellung I: Diese Frage wurde bereits zu wesentlichen Teilen in der Einleitung beantwortet. Das Ziel der Arbeit wurde mit der Entwicklung eines strukturbasierten Testansatzes für programmierte *Graphtransformationen (GTs)* bereits benannt. Die Korrektheit und ggf. die Angemessenheit der Implementierung sind die wesentlichen zu überprüfenden Eigenschaften des Testens, welches sich vornehmlich auf die Analyse des als besonders fehleranfällig zu sehenden Aspekts der korrekten Beschreibung von geeigneten Mustern konzentrieren sollte. Vor allem der beschriebene *Requirement Pattern Coverage (RPC)* Ansatz trägt diesen Aspekten Rechnung.

Zu Fragestellung II: Die Frage nach dem Wie des Testens ist allgemein zu beantworten: Es werden hinreichend viele geeignete Eingabemodelle benötigt, die bestenfalls in ihrer Gesamtheit zu einem ausreichend hohen Überdeckungsgrad, bezogen auf ein verfügbares und als geeignet geltendes Überdeckungkonzept führen. Im Speziellen sollte hier das Überdeckungskonzept nicht funktionsbasiert sein und nicht auf der Überdeckung von Quellcode basieren. Somit fehlten zu Beginn der Arbeit geeignete Konzepte; diese Lücke konnte aber durch den Ansatz aus Kapite 7 geschlossen werden. Außerdem ist ein möglichst vollständig automatisiertes (ggf. partielles) Orakel vorteilhaft, das deterministisch und sicher die Testergebnisse bewerten kann. Fehlbewertungen sollten weitestgehend ausgeschlossen sein, da ansonsten die Validität der Testergebnisse darunter leidet und ggf. zeitaufwendige, manuelle Nachkontrollen notwendig werden.

Zu Fragestellung III: Bezogen auf eine Wiederverwendung bestehender Ansätze sind verschiedene Aspekte zu betrachten. Funktionsbasierte Überdeckungskriterien (z. B. Metamodell- oder Constraint-basierte Verfahren) ließen sich auch in dem hier betrachteten Szenario unmittelbar anwenden (ggf. müssten dazu noch entsprechende Implementierungen und Werkzeuge so angepasst werden, dass eine direkte und automatisierte Verwendung möglich ist).

Bezogen auf strukturelle Kriterien waren existierende Ansätze für imperative Sprachen oder Kontrollflussgraphen nur sehr begrenzt geeignet, die betrachteten Systeme ausreichend gründlich zu testen, da der Aspekt der Mustersuche hierbei völlig unbeachtet bleiben würde. Kriterien für spezifische Modelltransformationssprachen sind, mit Ausnahme vielleicht des Ansatzes aus [Gei11] (dessen direkte Wiederverwendung hier durch technisch bedingte Inkompatibilitäten auszuschließen war), dagegen ungeeignet für eine Wiederverwendung, da sie sich nicht unmittelbar übertragen lassen.

Das hier entwickelte *RPC*-Konzept weist, je nach Betrachtungswinkel, entfernte Ähnlichkeiten zu traditionellen Überdeckungskonzepten auf. So gibt es gewisse Parallelen zum partitionsbasierten Testen, zum Testen auf Basis von Vorbedingungen

und es besteht, aufgrund des Austauschens von *Object-Variable (OV)*-Typen bzw. *Link-Variable (LV)*-Typen anhand der Informationen aus dem Metamodell bei der *Requirement-Pattern (RP)*-Generierung, auch eine gewisse Ähnlichkeit zu metamodellbasierten Überdeckungskriterien. Die Forderung nach der Überdeckung der Situation mit einem vorhandenen Treffer bzw. keinem vorhandenen Treffer pro *RP* könnte man ggf. als eine einfache Form der Bedingungsüberdeckung interpretieren. Die Ableitung der *RPs* selbst verläuft auf technischer Ebene nicht unähnlich zur Mutation, auch wenn sich die Verwendungen der jeweiligen Ergebnisse grundlegend unterscheiden. Aufgrund der Verwendung der unmodifizierten *Left Hand Side (LHS)* jeder Regel als ein eigenes *RP* subsummiert das *RPC*-Kriterium, unter Vernachlässigung von Sonderfällen wie Statement-Knoten, die Knoten- und Kantenüberdeckung des Kontrollflussgraphen.

Der zur Evaluation genutzte Mutationsansatz ist konzeptuell nicht neu, sondern wurde wiederverwendet. Neu ist dagegen die Menge problemspezifischer, angepasster Mutatoren und die Implementierung im Kontext von eMoflon.

Für das in der Arbeit eher als unabhängig und orthogonal betrachtete Problem der Testdatengenerierung lassen sich selbstverständlich existierende Ansätze wiederverwenden. In diesem Zusammenhang bietet sich allerdings ggf. noch Raum für weitergehende Forschung, wie im Folgenden noch genauer dargelegt.

Zu Fragestellung IV: Als direkte Antwort auf die Frage nach einem geeigneten Überdeckungsmaß kann die beschriebene Entwicklung von *RPC* verstanden werden. Dabei erfüllt *RPC* die wesentlichen gestellten Anforderungen: es ist unabhängig vom Quellcode, relativ leicht zu verstehen und basiert selbst auf der Mustersuche des *GT*-Ansatzes. Er bewegt sich damit auf der gleichen Abstraktionsebene wie das zu testende Transformationsproblem. Außerdem deuten die Ergebnisse aus den hier vorgestellten Experimenten darauf hin, dass das Testkonzept effektiv ist. Eine Übertragung der Idee – der Definition einer Überdeckung anhand von Graphmustern – auch auf andere *GT*-Ansätze sollte grundsätzlich möglich sein.

Zu Fragestellung V: Die Frage nach einer technischen Umsetzung lässt sich hier durch Verweis auf die bewusst weitgehend technologieneutrale Beschreibung der prototypischen Implementierung beantworten. Dabei zeigte sich, dass die entstandene Realisierung als Plugin für Eclipse (und Erweiterung für eMoflon) sinnvoll und praktikabel ist. Der zum *RPC*-basierten Testen vorgeschlagene Arbeitsablauf erscheint, unter praktischen Gesichtspunkten betrachtet, gangbar, da er sich mit dem bewussten Instrumentierungsschritt (vor der eigentlichen Messung der Überdeckung) nicht von anderen (Quellcode-basierten) Verfahren unterscheidet. Die eigentliche Testausführung unterscheidet sich mit und ohne Instrumentierung nicht wesentlich voneinander. Der Malus der schlechteren Ausführungsgeschwindigkeit bei vorhandener Instrumentierung und *RPC*-Auswertung ist letztendlich unumgänglich, lässt sich aber durch Optimierung noch geringfügig reduzieren. Darüber hinaus ist die Implementierung in erster Linie als prototypisch anzusehen und an manchen Stellen sicherlich noch verbesserungswürdig. So wäre die Umsetzung der *RP*-Generierung in Form einer *Higher-Order Transformation (HOT)* mittelfristig sicherlich erstrebenswert.

Zu Fragestellung VI: Die Untersuchung der Anwendung der Überdeckungskonzepte in der Praxis erfolgte im Rahmen einer umfangreicheren Evaluation. Grundsätzlich erfüllen beide Hauptfunktionalitäten den ihnen zugedachten Zweck.

Zu Fragestellung VII: Zur Untersuchung der Fragestellung nach der Leistungsfähigkeit des *RPC*-Ansatzes konnte hier die Technik der Mutationsanalyse als geeignetes Mittel der Wahl identifiziert werden. Der hierfür notwendige Mutationsansatz und die entsprechende Werkzeugunterstützung konnten im Rahmen der Arbeit entwickelt und praktisch nutzbar implementiert werden. Die Ergebnisse erster Experimente auf Basis der Maschinerie zeigen die praktische Durchführbarkeit und ermöglichten hier die exemplarische Bewertung der *RP*-Überdeckung am konkreten Beispiel.

Zu Fragestellung VIII: Als Antwort auf die Frage nach typischen Fehlern wurden existierende Fehlermodelle für *MTs* identifiziert und analysiert sowie, aufbauend darauf, ein an die *Story Driven Modeling (SDM)* Sprache angepasstes Fehlermodell entwickelt und vorgestellt.

Zu Fragestellung IX: Die erfolgreiche Umsetzung des Mutationsrahmenwerks zeigt, dass eine als *HOT* umgesetzte Realisierung nicht nur möglich, sondern auch leistungsfähig genug ist, um ausreichend viele Mutanten aus einer gegebenen *SDM*-Beschreibung abzuleiten. Die im Rahmen der Experimente zu beobachtenden großen Laufzeiten des unmittelbaren Mutationsprozesses, hauptsächlich aufgrund der langwierigen Codegenerierung, sind nur in Teilen dem Mutationsansatz selbst zuzuschreiben. Zwar bietet die Implementierung noch Möglichkeiten zur Optimierung, vgl. Abschnitt 8.4.3, dafür kann allerdings auch die beobachtete Geschwindigkeit des Codegenerators als vergleichsweise gering angesehen werden, insbesondere im direkten Vergleich zu einem parallel zu dieser Arbeit entwickelten neuen Codegenerator, der wesentlich schneller arbeitet. Für eine kleine bis moderat große Anzahl an Mutanten ist die einmalige Wartezeit deshalb tolerabel.

Zu Fragestellung X: Die letzte und vielleicht wichtigste Fragestellung kann prinzipiell nicht abschließend beantwortet werden. Anhand der Beispieltransformation konnte hier allerdings gezeigt werden, dass der Ansatz grundsätzlich funktioniert. Außerdem deutet die Evaluation darauf hin, dass der Ansatz tatsächlich dabei helfen kann, Fehler bzw. Unklarheiten in der Transformationsspezifikation zu entdecken. Auch ein positiver Zusammenhang zwischen der *RP*-Überdeckung und der Mutationsüberdeckung kann angenommen werden. Eventuelle statistische Zusammenhänge zwischen *RPC* und verschiedenen Formen der Codeüberdeckung müssten noch genauer untersucht werden, insbesondere dann, wenn sich die Struktur des Generats signifikant ändert (z. B. durch besagten neuen Codegenerator). Dies legt nahe, dass *RPC* zur effizienten Abschätzung der Mutationsadäquatheit herangezogen werden kann, und dass sich letztere auf Basis von *RPC* effektiv steigern lässt. Allerdings sind alle hierzu getroffenen Aussagen auf die konkreten Experimente und Messungen bezogen. Weitere (empirische) Untersuchungen wären also sehr wünschenswert.

Darüber hinaus kann der Mutationsansatz als etabliert gelten. Er wird gemeinhin als äußerst wirkungsvoll angesehen, vorausgesetzt die Mutationen erfolgen sinnvoll – allerdings zu dem Preis eines extrem hohen Aufwands. Die Nutzung der Mutationsüberdeckung zum kontinuierlichen Testen während der Entwicklung einer *MT* erscheint nach der Evaluation als eher unrealistisch.

Die wesentlichen, in dieser Arbeit enthaltenen Ergebnisse noch einmal kompakt zusammengefasst:

- Entwicklung der *RP*-Überdeckung,
- Implementierung der *RP*-Generierung, Automatisierung der Instrumentierung der bestehenden Transformation, Entwicklung einer Komponente zur Erfassung und Visualisierung der Überdeckung,
- Notwendigkeit eines Mutationstestansatzes zur Evaluation erkannt,
- Entwicklung eines Fehlermodells für *SDM*,
- Implementierung einer Menge konkreter Mutatoren als *HOT*,
- Mutationsrahmenwerk zur automatisierten Erzeugung von mutierten Transformationsbeschreibungen sowie zur Testausführung und –auswertung,
- Praktische Erprobung der Implementierungen an einer umfangreichen Beispieltransformation, die separat von dieser Arbeit entwickelt wurde,
- Evaluation von *RPC* anhand der Beispieltransformation sowie den entwickelten Testmaschinerien.

10.2 Anwendbar- und Übertragbarkeit

Zusammenfassend ist für die Ergebnisse dieser Arbeit zu sagen, dass die verwendeten und vorgestellten Konzepte wie beispielsweise (i) die Definition eines Abdeckungskriteriums über generierte Graphmuster, (ii) die Instrumentierung der Implementierung durch Anweisungen zur Auswertung der Überdeckung, (iii) die Strategien zur Konstruktion von *RPs* anhand der ursprünglichen Regel, (iv) das mutationsbasierte Testen und die Umsetzung als *HOT* usw. losgelöst von der konkreten Technologie zu sehen sind und somit eine Übertragung der Konzepte auf andere *GT*- bzw. *MT*-Ansätze grundsätzlich nicht ausgeschlossen ist.

Die Entwicklung sowie die Umsetzung der Methodik erfolgten allerdings mit dem konkreten Ziel, speziell *SDM*-Transformationen systematisch testen zu können. Bei einem strukturbasierten Überdeckungskriterium ist eine solche enge Anpassung an die Zielsprache nicht ungewöhnlich. Somit sollte es niemanden verwundern, dass die Implementierung *SDM*- und sogar eMoflon-spezifisch ist.

Darüber hinaus ist das eMoflon-Werkzeug im Rahmen der universitären Forschung sowie der sonstigen Pflege und Weiterentwicklung einem Prozess steter Anpassungen und Änderungen unterworfen. Das führt dazu, dass für das vorgestellte Rahmenwerk noch einige der parallel stattgefundenen Entwicklungen nachvollzogen werden müssten, um wieder dem aktuellsten Stand zu entsprechen. Auf *konzeptueller* Ebene ist diesbezüglich mit keinen bzw. vernachlässigbaren Schwierigkeiten zu rechnen. Größere Änderungen sind allerdings im Hinblick auf technische Details zu erwarten, wie sie sich beispielsweise aufgrund einer veränderten und optimierten Art der Serialisierung von *SDM*-Diagrammen ergeben. Grundsätzlich lassen sich die Auswirkungen solcher Änderungen durch konsequente und saubere Trennung der Zuständigkeiten zwischen verschiedenen Komponenten

reduzieren, so dass Änderungen vollständig transparent erfolgen können. Allerdings liegt dieser Idealfall hier (noch) nicht vor, da im Rahmen der prototypischen Implementierung des *RPC*-Ansatzes einige der zum Zeitpunkt der Erstellung benötigten „Low-Level"-Funktionalitäten unmittelbar und unabhängig implementiert werden mussten (was im Laufe der Zeit zu einem Auseinanderdriften führt). Das Nachziehen entsprechend notwendiger Anpassungen erscheint dennoch nicht unrealistisch. Unter Umständen wäre aber auch gleich eine modellbasierte Neuimplementierung anzustreben.

Für das Mutationsrahmenwerk ist dagegen mit weniger Anpassungsbedarf zu rechnen, da die Implementierung der Kernfunktionalität mit der *SDM*-Sprache selbst erfolgte und sowohl das *SDM*- als auch das *Eclipse Modeling Framework (EMF)*-Metamodell als sehr stabil anzusehen sind. Der Bedarf an Anpassungen, der sich aufgrund von Modifikationen an eMoflon ergibt, ist im Idealfall verschwindend gering bzw. mit einem erneuten Codegenerieren komplett verschwunden. Eventuell enthalten die *SDM*-basierten Implementierungen der Mutationen noch vereinzelt Modellierungsarten, die durch neue und ggf. striktere statische Analysen ausgeschlossen werden, so dass ggf. kleinere Anpassungen an den Diagrammen notwendig sein können. Natürlich wäre es, über die statischen Analysen hinaus, auch reizvoll, die Implementierung der Mutationstransformationen ausgiebig auf Basis der *RP*-Überdeckung zu testen.

10.3 Offene Punkte und Ausblick

Da der Umfang jeder Arbeit limitiert ist, müssen inhaltliche Grenzen gezogen werden. Manches muss folglich offen bleiben.[1] Im Folgenden werden einige der offenen Punkte, nach Themenkomplex sortiert, aufgegriffen. Den Schluss bilden Hinweise zu potentiell interessanten Fragestellungen für weitergehende Forschungsvorhaben.

10.3.1 RPC

Auf konzeptioneller Ebene ergeben sich mehrere Punkte, die in Bezug auf *RPC* noch untersucht werden könnten und ggf. auch sollten, um den Ansatz weiter zu verbessern. Folgende Aspekte könnten diesbezüglich von Interesse sein:

- Die Menge der Strategien zur Erzeugung der *RPs* ist als initial anzusehen. Es besteht grundsätzlich die Möglichkeit, weitere Strategien hinzuzufügen. Die Motivation für einzelne Strategien könnte beispielsweise aus der praktischen Erfahrung im täglichen Einsatz entstehen.
- Bisher umfassen die Prädikate, die aus den *RPs* gebildet werden, jeweils nur ein einzelnes, isoliertes Muster. Es sollten auch Prädikate mit komplexerer innerer Struktur (unter Zuhilfenahme des Kontrollflusses) ermöglicht werden. (Hierfür sind *SDM*-Schablonen, i. S. v. Templates, vorteilhaft.) Auch sind Verknüpfungen der bisher schon generierten Prädikate zu zusammengesetzten Aussagen denkbar, was allerdings voraussetzet, dass sich mehrere (eigentlich unabhängige) Auswertungsvorgänge auf weitestgehend identische Modellbestandteile beziehen. Dies führt direkt zum nächsten Punkt.

[1] Insbesondere konnten beispielsweise nicht sämtliche Veröffentlichungen, an denen ich im Rahmen meiner Dissertation mitgearbeitet habe, hier in vollem Maße Berücksichtigung finden.

- Ggf. würde hierfür eine Datenstruktur bzw. ein (bisher nicht vorhandenes) Sprachkonstrukt zur expliziten Übergabe von unvollständigen Teil-Matches an Muster, Regeln oder andere *SDM*-Operationen vorteilhaft sein. Eine andere potentiell interessante *SDM*-Erweiterung wären *OVs* die als „semi-gebunden“ anzusehen wäre: gebunden, falls ein Wert bekannt ist, ansonsten ein normal zu suchender Knoten im Modell. Ziel wäre in beiden Fällen, eine bessere Kontrollierbarkeit durch besondere Sprachkonstrukte zu erlangen und dadurch ggf. das Testen zu erleichtern.
- Bisher werden alle *RPs* automatisiert aus der zugrunde liegenden Originalregel abgeleitet. Es fehlt allerdings bisher noch die Möglichkeit, vom Benutzer selbst spezifizierte Muster in den Prozess einzubringen. Gäbe es diese Möglichkeit, könnten Anwender selbst Situationen vorgeben, die aus ihrer jeweiligen (Experten-)Sichtweise heraus für das Testen relevant sind. Setzt man diesen Gedankengang fort, wäre eine auf Graphmustern aufbauende Erweiterung der *SDM*-Sprache dergestalt möglich, dass für einzelne Regeln visuelle Vor- und optional auch Nachbedingungen (erstere im Sinne von Zusicherungen oder von Testanforderungen, letztere als Möglichkeit zur Implementierung partieller Orakel, vgl. hierzu z. B. auch [HL07; MC07]) spezifiziert werden könnten, deren Einhaltung dann dynamisch zur Laufzeit überprüft werden. Aufbauend darauf ließe sich dann ggf. auf Verletzungen von Zusicherungen geeignet reagieren. Auch wäre es hierdurch möglich, die sog. *Vigilance* [LBJ06] der Transformation zu erhöhen, also ihre Fähigkeit, Fehlerzustände während der Laufzeit zu erkennen.
- Es besteht potentiell ein Berührungspunkt des Testverfahrens mit Verfahren zur formalen Verifikation für *GTs*. Testanforderungen könnten beispielsweise hinsichtlich ihrer generellen Erfüllbarkeit (unter Bezugnahme auf das (oder die) Metamodell(e) und unter Vernachlässigung von Einschränkungen durch den Kontrollfluss sowie vorausgehenden Regeln) hin überprüft werden und von vorn herein unerfüllbare Anforderungen von der Betrachtung ausgeschlossen werden. Umgekehrt könnten manuell als nicht erfüllbar identifizierte Anforderungen hinsichtlich ihrer grundsätzlichen Erfüllbarkeit überprüft werden. Ist eine ausgeschlossene Anforderung grundsätzlich erfüllbar, liegt ggf. eine fehlerhafte Klassifizierung vor. Auch bestehende Konstruktionen zur Umwandlung von Nach- in Vorbedingungen könnten u. U. genutzt werden, um schrittweise die Vorbedingungen einer Regel (irgendwo „in der Mitte“ des Kontrollflusses) so lange schrittweise in Richtung Startknoten zu verschieben, bis sich die Anforderungen an eine entsprechende Testeingabe unmittelbar ablesen lassen. Vgl. hierzu den nächsten Punkt.
- Eine Testgenerierung auf Basis von *RPC*, z. B. mittels symbolischer Ausführung und Model-Finding.

Bezüglich der Implementierung ergeben sich selbstverständlich, alleine schon aufgrund des prototypischen Charakters, Möglichkeiten zur Verbesserung. Hier werden nur einige Punkte exemplarisch aufgegriffen:

- Die Implementierung insbesondere der eigentlichen *RP*-Generierung sollte an die aktuelle eMoflon-Version angepasst werden. Ggf. ist dafür auch eine modellbasierte Neuimplementierung auf *SDM*-Basis anzustreben – auch weil diese selbst *RP*-getrieben getestet werden kann.
- Die *SDM*-Sprache und/oder der Codegenerator könnten so angepasst werden, dass der *RPC*-Ansatz möglichst gut unterstützt wird, z. B. durch Spracherweiterungen wie die der Unterstützung visueller Vor- und Nachbedingungen, parametrisierbarer

Muster, zu Letzterem vgl. z. B. [Leg13], oder angepasster Optimierungen bzgl. der *RP*-Auswertung (viele Pattern-Matching-Schritte wiederholen sich bei den einzelnen *RPs*).
- Bisher fehlt eine Möglichkeit zur Benutzerinteraktion, was die Auswahl der tatsächlich genutzten *RP*-Generierungsstrategien angeht. Die Implementierung könnte dahingehend erweitert werden, dass nur bestimmte *RPs* generiert und ausgewertet werden.
- Auch ein benutzergesteuertes Ausschließen einzelner *RPs* von einer weiteren Berücksichtigung erscheint sinnvoll, z. B. um unerfüllbare Anforderungen auszusortieren.
- Die Stabilität und die Robustheit des gesamten Instrumentierungsprozesses könnte noch verbessert werden. Ein möglichst breiter Einsatz würde eventuelle Schwächen sicherlich aufdecken helfen.
- Bisher werden die generierten *RPs* mit Hilfe der Visualisierungskomponente von Fujaba4Eclipse angezeigt. Umstellungen am Codegenerierungsprozess (sowie die Tatsache, dass dieses Projekt zurzeit nicht mehr aktiv gepflegt wird) erfordern eine Umstellung auf eine andere Art der Präsentation. Anzustreben wäre eine Visualisierung in dem Editor, in dem die *SDM*-Diagramme originär erstellt werden.

10.3.2 Mutationsrahmenwerk

Konzeptionell ließe sich das Mutationsrahmenwerk dahingehend erweitern, dass zusätzliche Mutationsarten realisiert werden. Dazu müssten eventuell erst neue Fehlerarten identifiziert werden. Auf jeden Fall wären neue Mutatoren zu entwickeln und zu implementieren. Letzteres ist unter Ausnutzung des bestehenden Rahmenwerks relativ leicht möglich. Um potentiell sinnvollen Mutationen nachzuspüren, könnten, wie in [Pil14] skizziert, *SDM*-Anwender (oder Anwendergruppen, beispielsweise unterteilt nach ihrer Qualifikation) systematisch nach ihren Erfahrungen befragt werden.

Eine andere Richtung, in der das Mutationsrahmenwerk verbessert werden könnte, besteht in einem eingehenden Vergleich der verschiedenen Mutatoren im Hinblick auf ihre Leistungsfähigkeit und den durch sie bedingten Aufwand. Ammann und Offutt merken in [AO08, S. 182] an, dass ggf. schon einige wenige besonders geeignete Mutatoren, die sog. *effektiven* Mutatoren, ausreichen, um sehr gute Resultate zu erhalten. Dies hätte den Vorteil, dass die Menge der Mutanten und damit sowohl die Anzahl der auszuführenden Testläufe als auch der notwendige Aufwand zur Analyse der Resultate reduziert wird. Für traditionelle Programmiersprachen wurden solche Zusammenstellungen bereits (empirisch) untersucht, vgl. ebenfalls [AO08, S. 182]. Im Kontext der Mutation von Modelltransformationen und insbesondere von Graphtransformationen fehlen entsprechende Untersuchungen bisher weitestgehend.

Eine weitere Optimierung läge ggf. in einer dedizierten Unterscheidung der beiden typischen Anwendungsszenarien der Mutation (Analyse und Testen). Die Anforderungen für die Durchführung einer Mutationsanalyse, wie z. B. eine große Anzahl an Mutanten, möglichst gleichverteilt über die gesamte Implementierung des *System Under Test (SUT)*, unterscheiden sich von denjenigen, die für die Durchführung des mutationsbasierten Testens benötigt werden, wie z. B. möglichst kleine Menge an Mutanten für eine schnelle Testausführung, ggf. lokal begrenzte Mutation für den gerade modifizierten und zu testenden Bereich, schnelles und komfortables Neugenerieren von Mutanten nach Än-

derungen am *SUT*. Das Mutationsrahmenwerk wurde vor allem für die Mutationsanalyse mit dem Ziel der empirischen Bewertung eines anderen Testverfahrens entwickelt. Der zweite Aspekt könnte also noch ausgebaut werden, um so auch im direkten Vergleich zu *RPC* als weniger schwergewichtig zu erscheinen. Auch könnte man ggf. die Optimierung/Minimierung einer Testmenge durch das Identifizieren und Entfernen redundanter Tests in einem dritten Anwendungsfall mehr in den Fokus rücken.

Aus praktischer Sicht wäre in dem konkreten Aufbau auch eine Untersuchung der Kombinierbarkeit der Mutationen mit zufallsbasierten Techniken wie der *Feedback-directed Random Test Generation* [Pac+07] interessant. Hiermit böte sich u. U. eine effektive und effiziente Möglichkeit zur Erweiterung der Testmenge um adäquate Tests.

Im Hinblick auf den Aspekt einer konkreten Implementierung ist auch hier auf den prototypischen Zustand der Umsetzung hinzuweisen. Allerdings konnte bereits aufgrund der Erfahrung aus der Realisierung der *RP*-Überdeckung die Implementierung des Mutationsrahmenwerks profitieren. Die Wahl von *SDM* als Implementierungssprache hat sich beispielsweise als sehr günstig erwiesen, auch wenn einige technische Besonderheiten im Zusammenspiel mit *EMF* hierbei zu beachten waren, welche den Einsatz der *SDM*-basierten Entwicklung erschwert haben.

Aus Benutzersicht sind höchstwahrscheinlich die folgenden vier Aspekte besonders verbesserungswürdig.

- Insgesamt sollte für den praktischen Einsatz die Robustheit noch erhöht werden. So ist beispielsweise die Ausnahmebehandlung im Rahmen der Testausführung noch nicht optimal.
- Andererseits wäre es gut, wenn man von einer initialen Menge abgeleiteter Mutanten nicht immer gleich alle beim Testen nutzen müsste. Bisher gibt es nur den Ansatz, der als „ganz oder gar nicht“ zu bezeichnen ist. Zwar lassen sich zu Beginn die auszuführenden Mutationen auswählen und ggf. einschränken, die Menge der tatsächlich generierten Mutanten muss dann aber immer vollständig genutzt werden.
- Weiterhin existiert momentan keine Vorschau der durchzuführenden Mutationen. Somit fehlt dem Benutzer möglicherweise die notwendige Grundlage zur bewussten Auswahl durchzuführender Mutationen.
- Andererseits wurden in Abschnitt 8.4.3 bereits einige potentielle Optimierungen beschrieben, die eine Benutzung erleichtern würden. Aus Anwendersicht wäre sicherlich auch interessant zu wissen, mit welchem Geschwindigkeitsvorteil durch den Einsatz des neuen Codegenerators zu rechnen wäre. Dieser besitzt zumindest genügend Potenzial, um den in [AO08, S. 275] erwähnten *Compilation Bottleneck* deutlich zu entschärfen.

10.3.3 Messungen erweitern

Die Evaluation, obwohl bereits recht umfangreich, könnte selbstverständlich noch erweitert werden. Alle auf konkreten Beispielen beruhenden Aussagen und Erkenntnisse sind letztendlich nur als exemplarisch anzusehen. Somit sollte auch hier angemerkt werden, dass weitere Experimente und Messungen mit anderen Beispielen sinnvoll erscheinen. Insbesondere wäre aus wissenschaftlicher Sicht die Durchführung einer umfassenden Benutzerstudie im praktischen Einsatz interessant (auch wenn deren Ergebnisse dann ebenfalls empirisch wären). So könnte man beispielsweise drei (oder eine durch drei teil-

bare Anzahl von) Gruppen von Entwicklern jeweils die gleiche Transformationsaufgabe lösen lassen. Die erste Gruppe könnte dann ausschließlich klassische Testverfahren auf Quellcodeebene, die zweite Gruppe das *RPC*-Verfahren und die dritte Gruppe das Mutationstesten einsetzen. Auf Basis entsprechender Beobachtungen könnten dann z. B. der benötigte Zeitaufwand, die wahrgenommene Komplexität des Verfahrens und die Werkzeugunterstützung verglichen werden. Am interessantesten wäre allerdings der Vergleich der entstandenen Testmengen hinsichtlich der Erkennungsrate von Fehlern z. B. in einer vierten, unabhängigen Referenzimplementierung.

Hier wurde das Problem der Nichtdurchführbarkeit einer Studie durch synthetische Fehlererzeugung versucht zu umgehen, was als durchaus etablierter Ansatz gelten darf. Durch die Verwendung einer umfangreichen und fremden Transformation mit unabhängig erstellter initialer Testmenge wurde versucht, jedweden Eindruck von systematischer oder unbewusster Verzerrung auszuräumen. Die Vielzahl an Arbeiten zur Effektivität des Mutationstestens sowie zu Vergleichen unterschiedlicher Überdeckungskriterien zeigt, dass der sichere Nachweis von belastbaren Eigenschaften ein grundsätzliches Problem im Forschungsgebiet des (nichtvollständigen) Testens ist.

10.3.4 Potentielle zukünftige Entwicklungs- und Forschungsaufgaben

Während der Erstellung dieser Arbeit traten einige potentielle Fragestellungen und Forschungsgegenstände zu Tage, deren genauere Untersuchung eventuell als gewinnbringend einzuschätzen ist. Am Ende der Arbeit sollen die Wichtigsten noch kurz erwähnt werden.

Multifunktionaler Modellgenerator

Ein Thema mit hoher praktischer Relevanz, nicht nur für die Qualitätssicherung, stellt die Generierung von Instanzmodellen mit definierten Eigenschaften zu einem gegebenen Metamodell dar. Hierzu existieren bereits verschiedene Ansätze und Techniken sowie entsprechende Forschungsprototypen und Werkzeuge. Was allerdings, meiner Meinung und meinem aktuellen Kenntnisstand nach, noch fehlt, wäre ein integriertes Rahmenwerk oder Werkzeug, das einerseits alle wesentlichen Feinheiten von Ecore- bzw. *EMF*-basierten Metamodellen sowie Attribute und Bedingungen über diesen unterstützt und andererseits in der Lage ist, verschiedene Suchtechniken sinnvoll zu kombinieren oder zumindest als Alternativen anzubieten. Im Idealfall könnten dann zu suchende Modelle anhand der Angabe einer Menge von deklarativ formulierten Eigenschaften beschrieben oder teilweise vorgeben werden, vergleichbar wie dies Alloy [Jac02] in Ansätzen bereits bietet. Zusätzlich wären statistische Vorgaben oder Einschränkungen durch ein Feature-Modell denkbar. Ziel müsste es sein, zu untersuchen, inwiefern sich imperative Konstruktionsvorschriften (ASSL [GBR05]), auf Logik-basierten Beschreibungen aufbauende Model-Finding-Ansätzen (Alloy, EMFtoCSP [Gon+12], UML2Alloy [Ana+07], etc. egal ob auf SAT, CSP oder SMT basierend), graphgrammatikbasierte Verfahren, kombinatorische Verfahren (T-Wise-Überdeckung, Feature-Kombinationen) und heuristische Algorithmen (z. B. problemangepasste evolutionäre Algorithmen zur Ableitung neuer Lösungen aus bestehenden Lösungen) miteinander sinnvoll kombinieren ließen und welche positiven Effekte dadurch entstehen könnten.

Testen von und mit TGGs

Das eMoflon-Werkzeug unterstützt nicht nur Modelltransformationen auf Basis der *SDM*-Sprache sondern auch sog. *Triple Graph Grammar (TGG)* Spezifikationen. Für letztere wurden bereits Testüberdeckungskonzepte entwickelt, die auf der Abdeckung von Produktionen bzw. von Produktionssequenzen basieren. Darüber hinaus eignen sich *TGGs* grundsätzlich auch als Testmodell für das modellbasierte Testen, insbesondere bei bidirektionalen Transformationsaufgaben. Andererseits wäre es sicherlich auch bei *TGGs* von Interesse, sicherzustellen, dass eine konkrete, für eine einzelne *TGG*-Regel ableitbare Operationalisierung gründlich getestet wurde. Ein Testansatz müsste also nicht nur alle potentiell möglichen Regelsequenzen (bis zu einer gewissen Obergrenze) berücksichtigen, sondern auch noch beachten, dass sich je nach Anwendungsstelle – also je nach Wahl des Kontextes einzelner Produktionen – unterschiedliche Strukturen im Modell ergeben. Da die Anzahl der möglichen Zustände, in denen eine einzelne konkrete Operationalisierung zur Ausführung kommen kann, selbst bei überschaubar langen Regelanwendungssequenzen als Initialisierungsschritt sehr schnell anwächst und zu groß für das vollständige Testen wird, müssen einige repräsentative Zustände stellvertretend identifiziert werden. Hierzu wird ein geeignetes Überdeckungskonzept benötigt, das sich nicht alleine auf die Regeln oder deren Anwendungssequenzen stützt, sondern auch systematisch die Menge potentiell vorhandener (Teil-)Strukturen im Modell abdeckt. Letzteres könnte ggf. ebenfalls anhand einer zu *RPC* vergleichbaren Abdeckung anhand einer Art von Prädikaten auf Musterbasis erreicht werden. Des Weiteren müsste ein solcher Testansatz auf die Ausführung einer zu testenden Operationalisierung soweit Einfluss nehmen können, dass *alle potentiell möglichen* Anwendungsstellen (für einen bestimmten, konkreten Ausgangszustand) durchprobiert und so die jeweiligen Ausführungen getestet werden.

In diesem Zusammenhang könnte auch das Testen von inkrementellen Modellsynchronisationsschritten von Interesse sein. Denn letztere lassen sich auch aus einer *TGG*-Beschreibung ableiten [Sch95; Leb+14].

Erweiterung der Testrahmenwerks

Das hier vorgestellte Testrahmenwerk für *SDM*-Transformationen umfasst zurzeit keine unmittelbare Unterstützung für die Erstellung konkreter Testfälle oder für die konkrete Bewertung von Testausgaben. Weder das automatisierte bzw. manuelle Erzeugen von Testmodellen noch das Spezifizieren von partiellen oder vollständigen Orakeln sind bisher Gegenstand einer Werkzeugunterstützung. Ein Ausbau des Rahmenwerks, das bisher im Wesentlichen der Bestimmung der Testüberdeckung und dem Testen mit Mutanten dient, zu einem vollständig integrierten Ansatz zur Testerstellung, -durchführung und -auswertung wäre aus Anwendersicht wünschenswert.

Datenflussbasiertes Testen und GTs

Für das Testen traditionell erstellter Software stellen datenflussbasierte Testkriterien [FW88] eine leistungsfähige Alternative zu den verbreiteteren Verfahren dar. Insbesondere für Graphtransformationen fehlen entsprechende Konzepte bisher weitestgehend. Eine gewisse Ausnahme stellen Arbeiten von Heckel und Khan et al. dar [HKM11; KRH12a], die Beziehungen zwischen Paaren von *GT*-Regeln und verschiedene Formen der Abdeckungen bezogen auf Regelanwendungssequenzen betrachten. Ggf. ist dieses Feld aus

Sicht der Forschung noch ergiebiger und entsprechend interessant genug, um weitergehende Arbeiten zu motivieren. Ein Gedanke, den beispielsweise auch die erwähnten Autoren im Rahmen ihres Ausblicks in [HKM11] aufgreifen.

Offenes Test- und Vergleichsrahmenwerk

Eine weitere Richtung für zukünftige Entwicklungsvorhaben könnte in der Erstellung einer offenen, standardisierten und erweiterbaren Test- und Vergleichsumgebung für *MTs* im EMF-Umfeld liegen. Ein ganz ähnliches Ziel wird beispielsweise in [GS13] für den *trans*ML-Ansatz verfolgt. Ziel einer solchen Umgebung wäre es, beliebige Implementierungen einer ausreichend genau spezifizierten MT-Aufgabe entgegenzunehmen – dazu müsste die Lösung einer gewissen Schnittstelle genügen – diesen eine standardisierte Laufzeitumgebung zur Verfügung zu stellen und sie so kontrolliert ausführbar zu machen. So wäre es möglich, unterschiedliche Lösungen unter kontrollierten und einheitlichen Bedingungen beispielsweise in Bezug auf ihre jeweilige Korrektheit hin zu überprüfen. Zu testende Konstellationen könnten dazu beispielsweise vorgegeben werden oder anhand von Testspezifikationen, z. B. in Form der zuvor erwähnten TRACTs, beschrieben werden. Auch eine automatisierte Generierung von Eingabedaten (anhand funktionaler Abdeckungskriterien) könnte dazu unterstützt werden. Eine solche Umgebung sollte im Idealfall auch den einfachen Wechsel des genutzten Orakels ermöglichen. Standardmäßig könnte Letzteres beispielsweise aus einem Vergleich zwischen der Ausgabe des *SUT* mit der Ausgabe einer Referenzimplementierung bestehen.

Zum Schluss: Im Rahmen der Arbeit konnten einige Erkenntnisse gewonnen werden. Auch sind diverse daran anschließende Fragestellungen offenkundig geworden, die für weitergehende Untersuchungen interessant und relevant erscheinen.

Anhang

A Implementierung der Beispieltransformation

Dieser Abschnitt beinhaltet die vollständige Implementierung der BlockDiagram-To-SimpleJava-Transformation (Bd2Ja). In den Unterabschnitten A.1 und A.2 sind die Metamodelle der beiden Teilsprachen gegeben. Die Transformation wird in Abschnitt A.3 beschrieben. Für die eigentlichen *SDM*-Diagramme siehe Unterabschnitt A.3.2 bzw. A.3.3.

A.1 Die Block-Diagram-Sprache

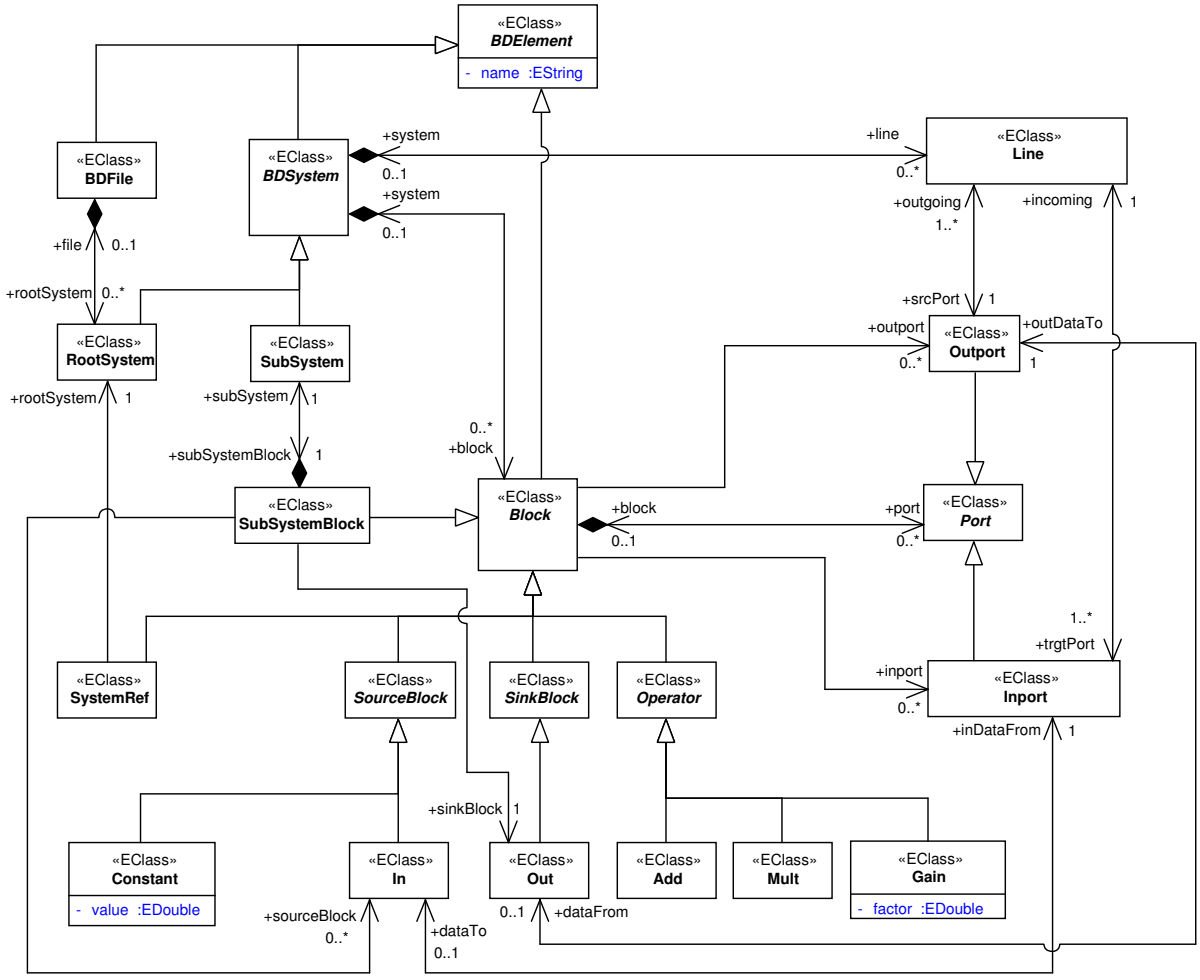


Abbildung A.1: Blockdiagramm-Metamodell

A.2 Die SimpleJava-Sprache

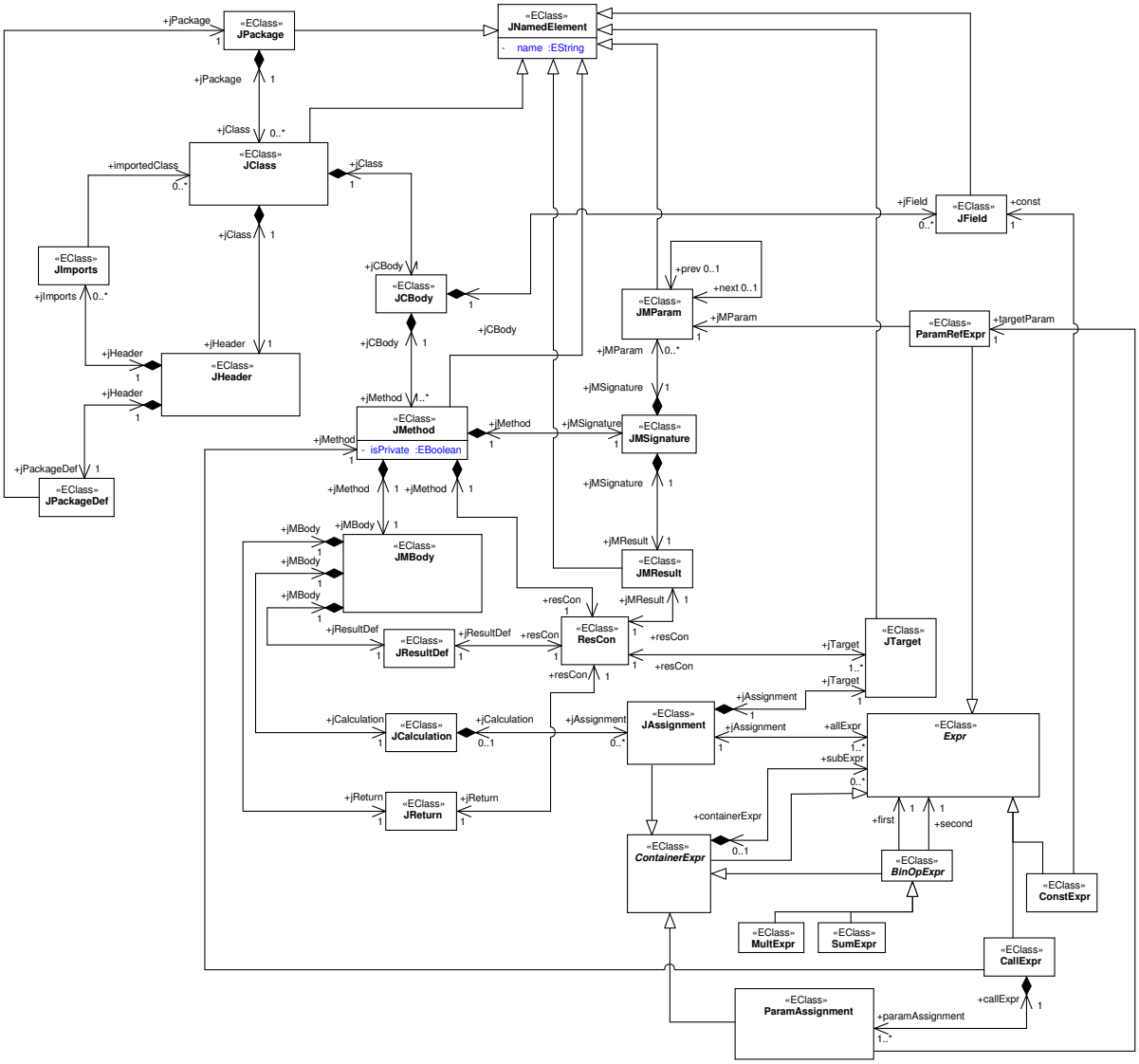


Abbildung A.2: Metamodell für eine stark vereinfachte Teilmenge von Java

A.3 Die Transformation

A.3.1 Das Bd2Ja-Transformationsmetamodell

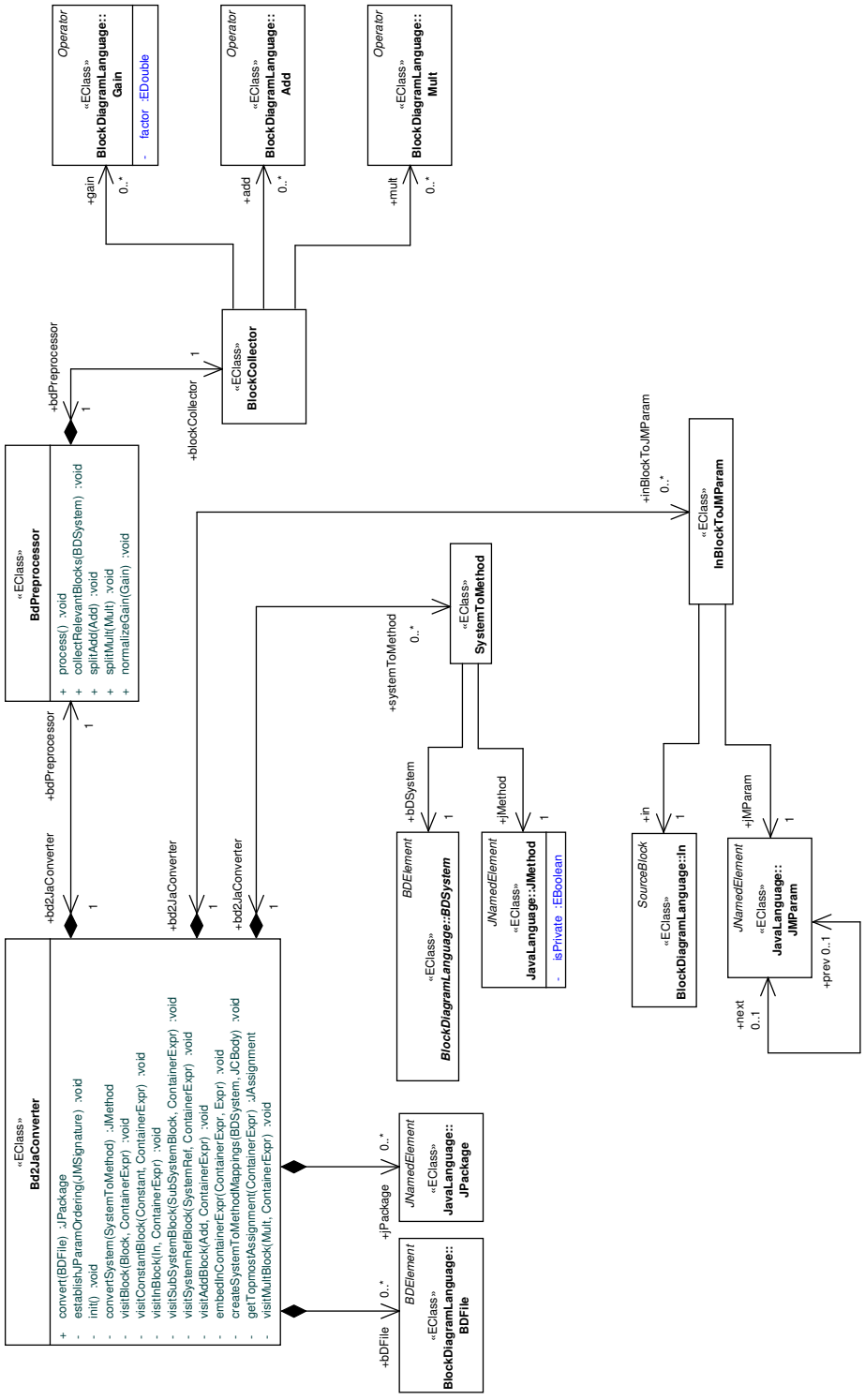


Abbildung A.3: Transformationsmetamodell

A.3.2 Normalisierung

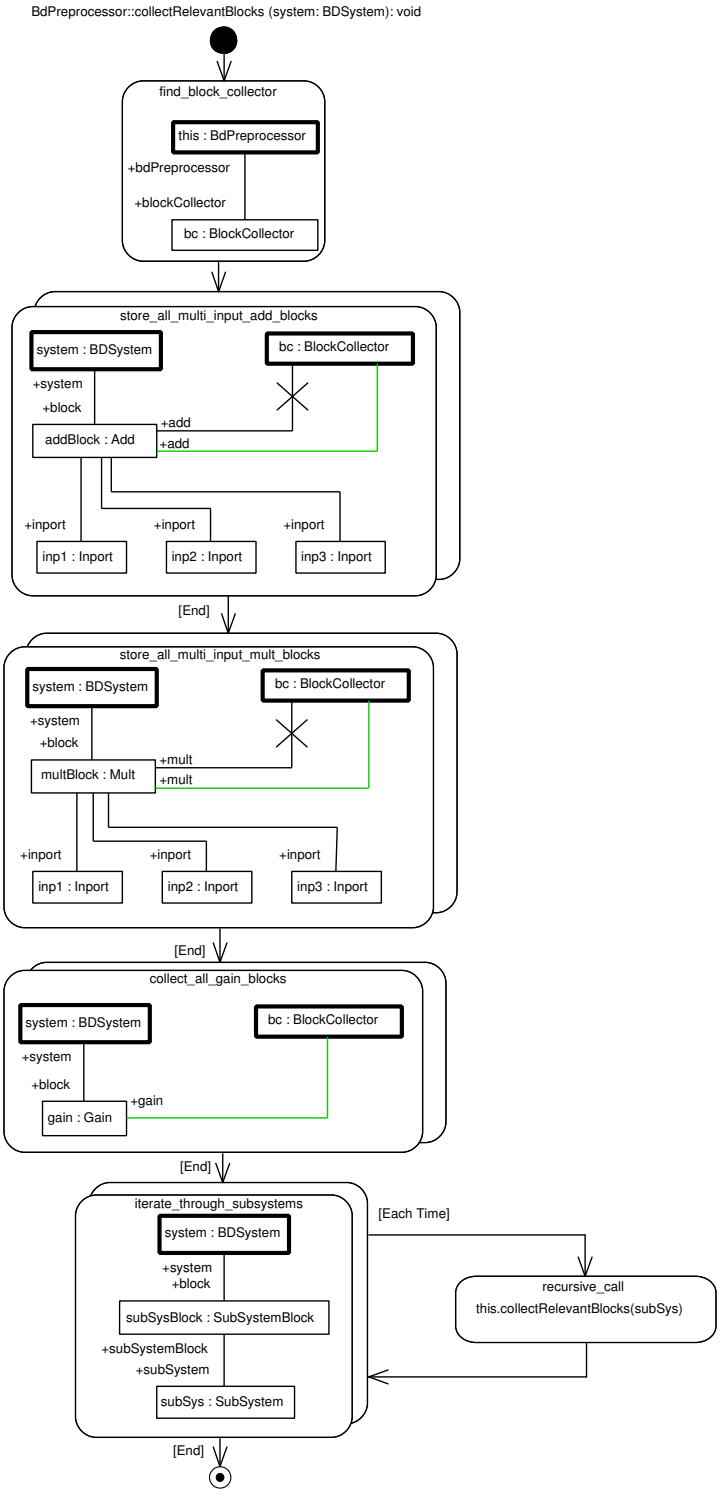


Abbildung A.4: `BdPreprocessor::collectRelevantBlocks(BDSystem)`

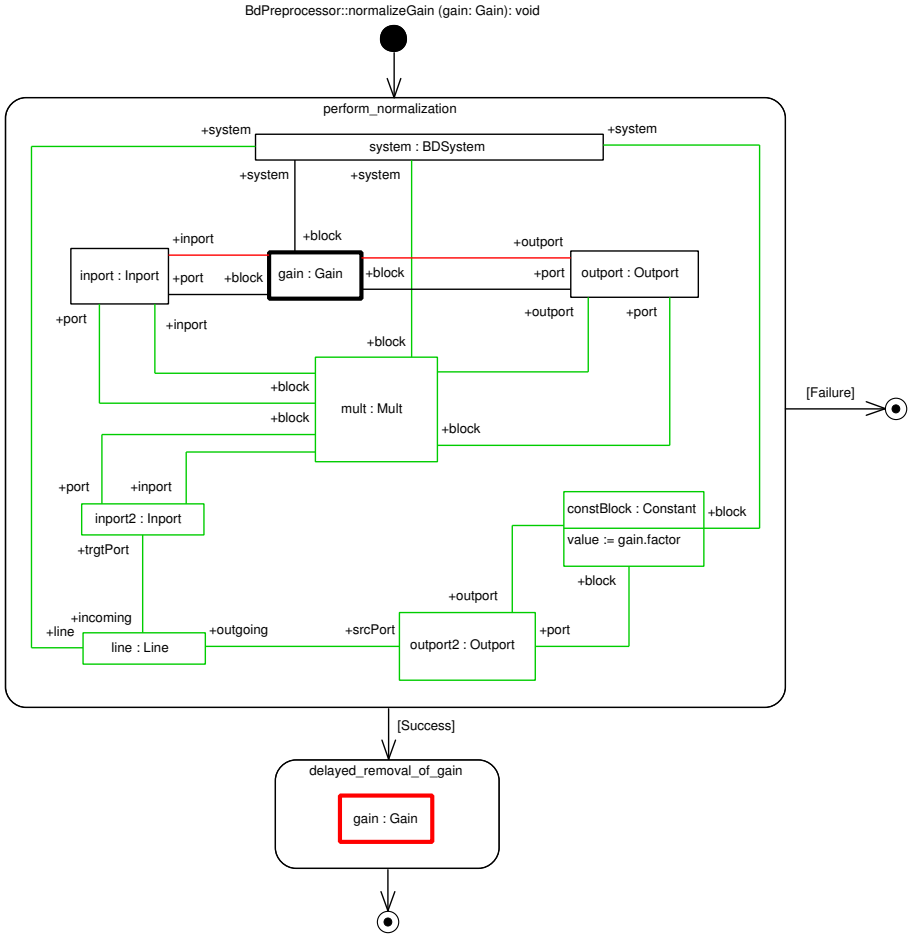


Abbildung A.5: `BdPreprocessor::normalizeGain(Gain)`

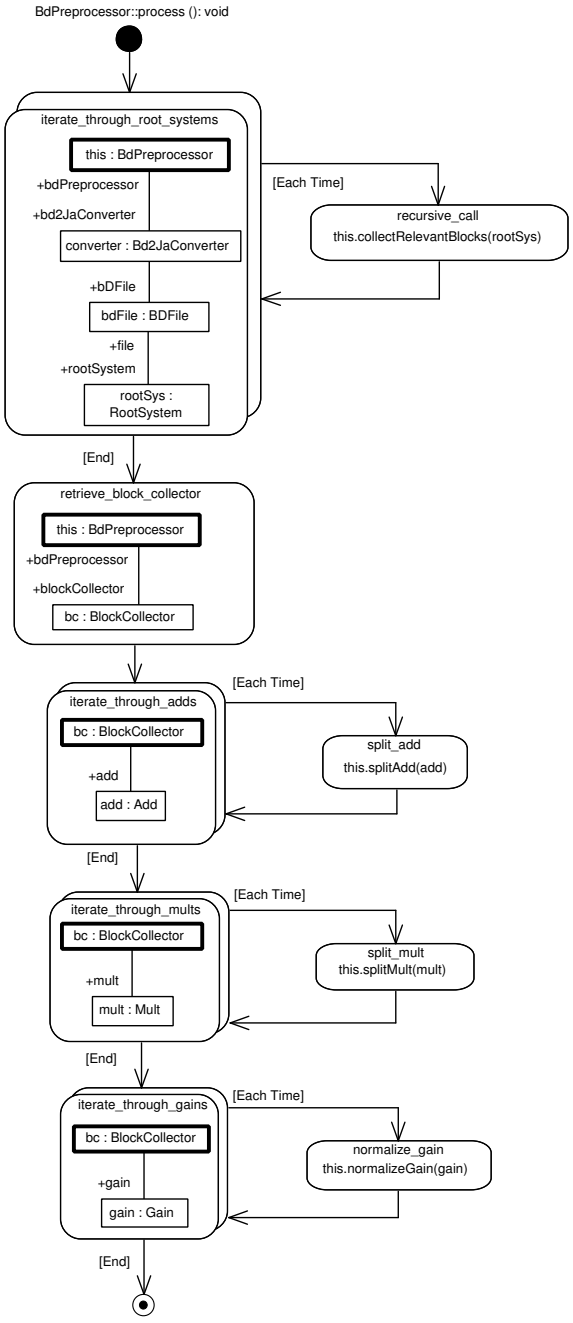


Abbildung A.6: `BdPreprocessor::process()`

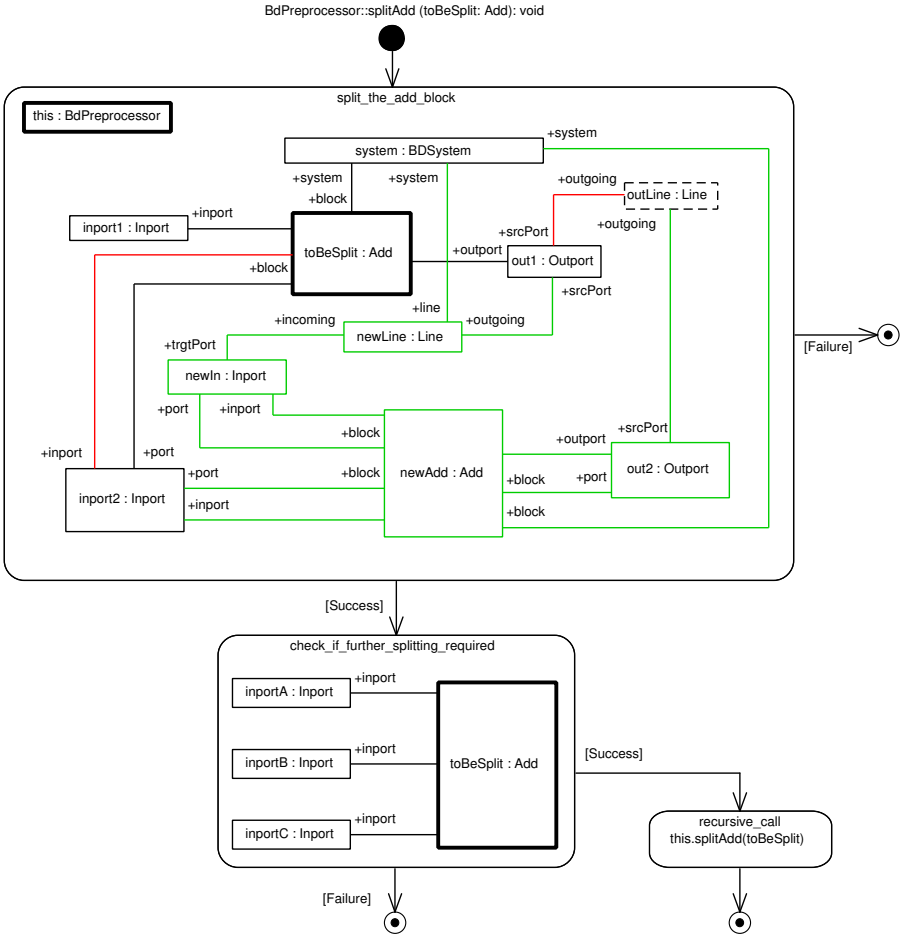


Abbildung A.7: `BdPreprocessor::splitAdd(Add)`

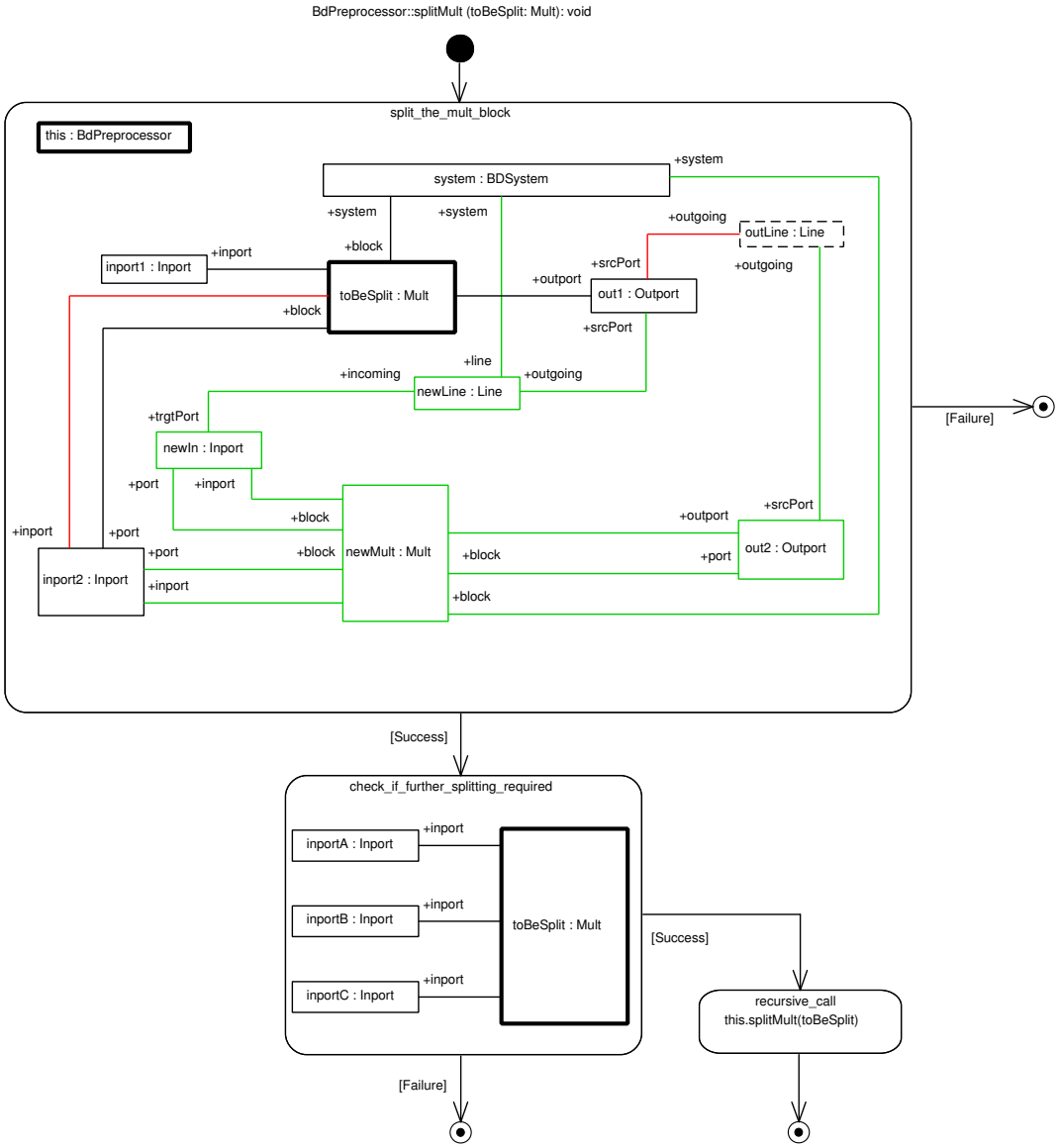


Abbildung A.8: `BdPreprocessor::splitMult(Mult)`

A.3.3 Übersetzung

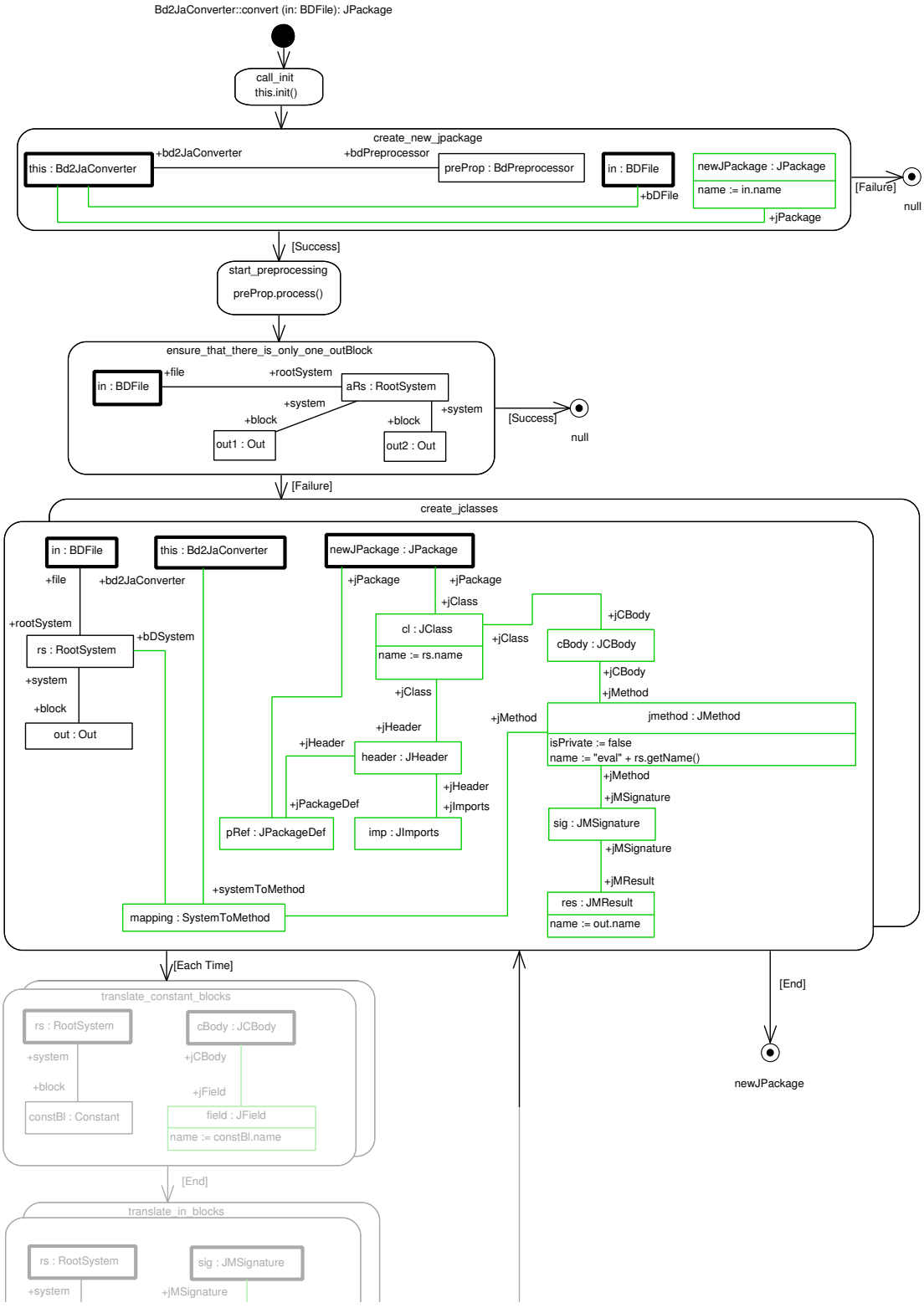


Abbildung A.9: `Bd2JaConverter::convert(BDFile)` - Teil 1 von 2

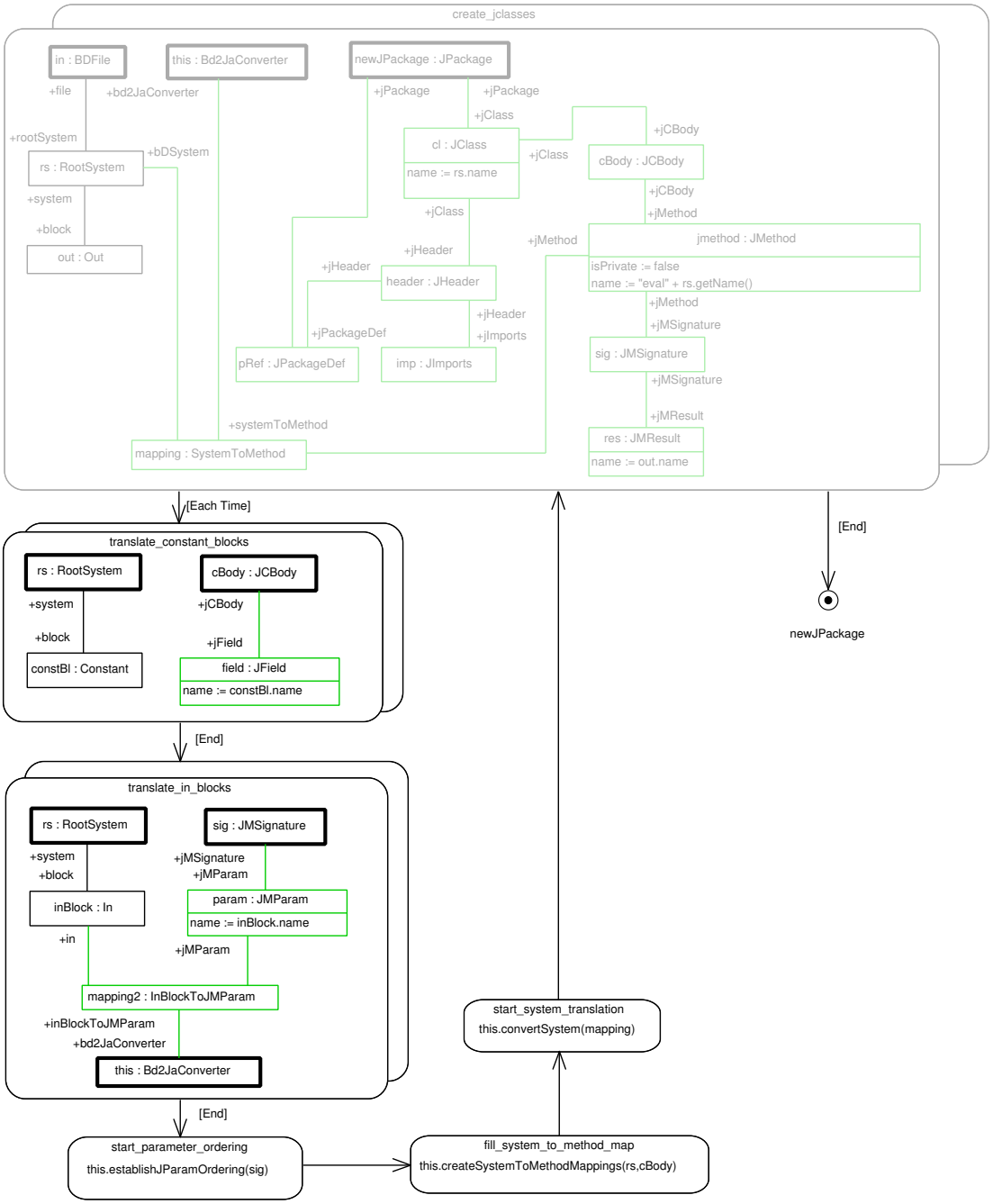


Abbildung A.10: `Bd2JaConverter::convert(BDFile)` - Teil 2 von 2

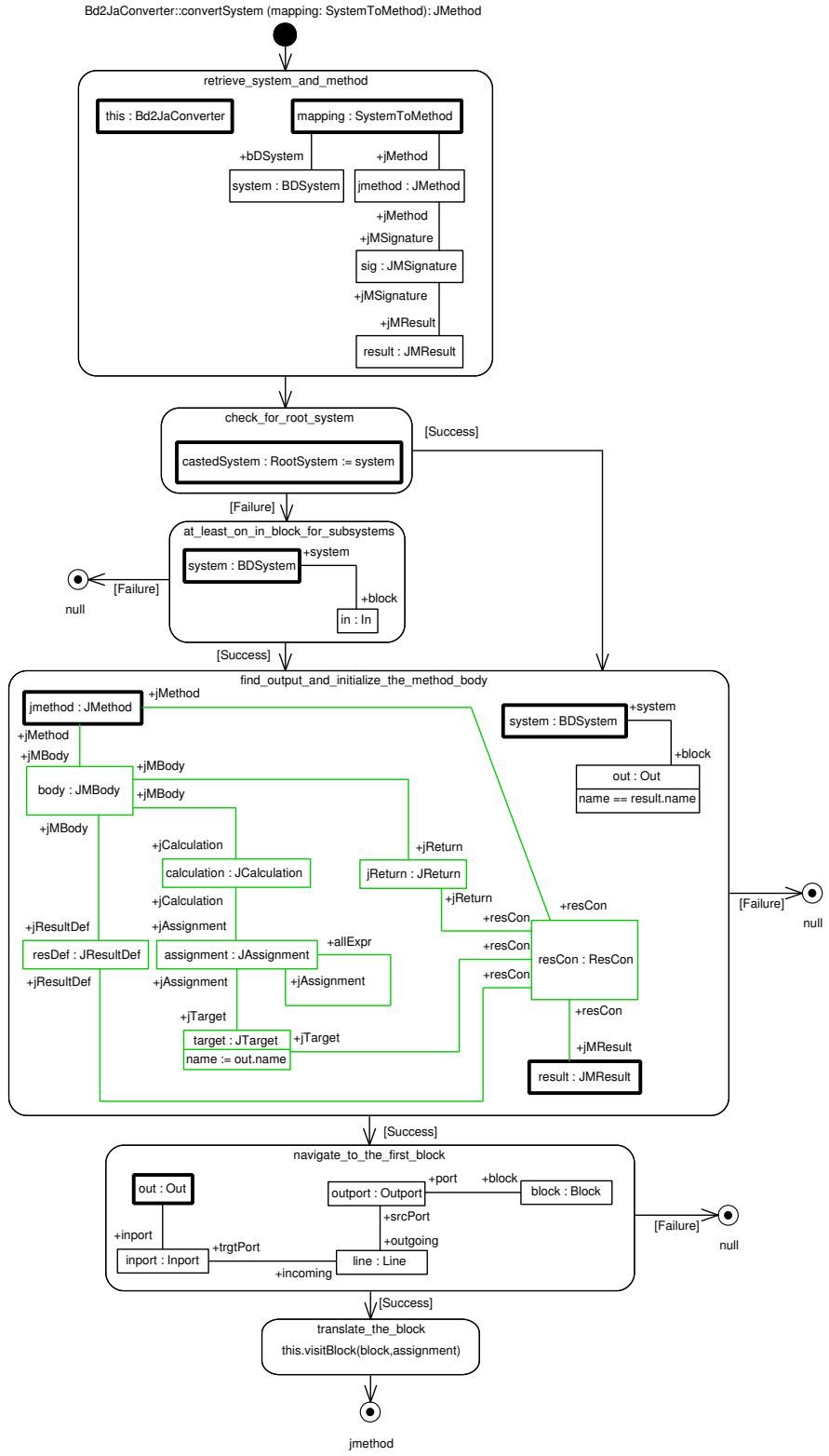


Abbildung A.11: `Bd2JaConverter::convertSystem(SystemToMethod)`

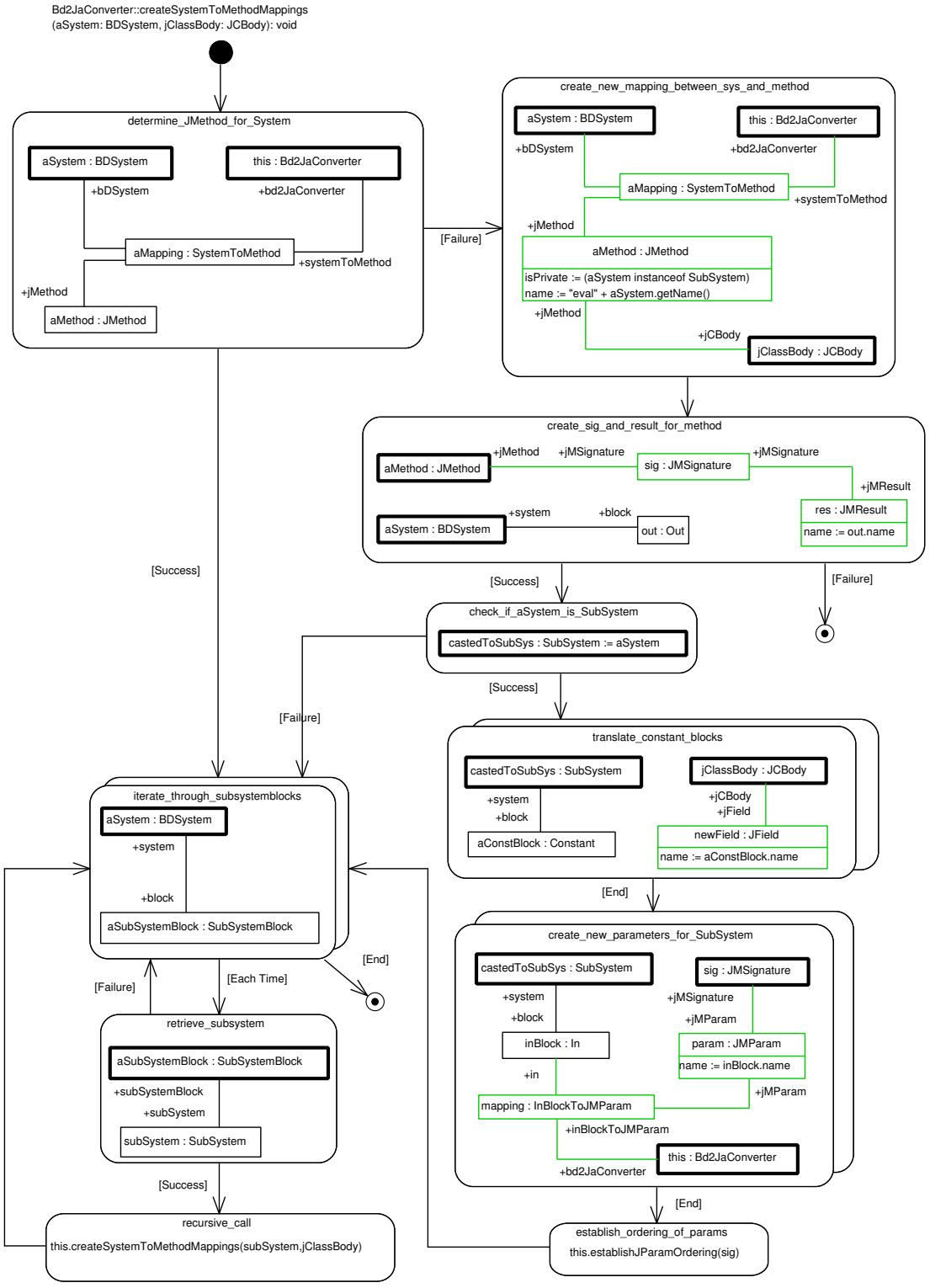


Abbildung A.12: `Bd2JaConverter::createSystemToMethodMappings(BDSystem, JCBody)`

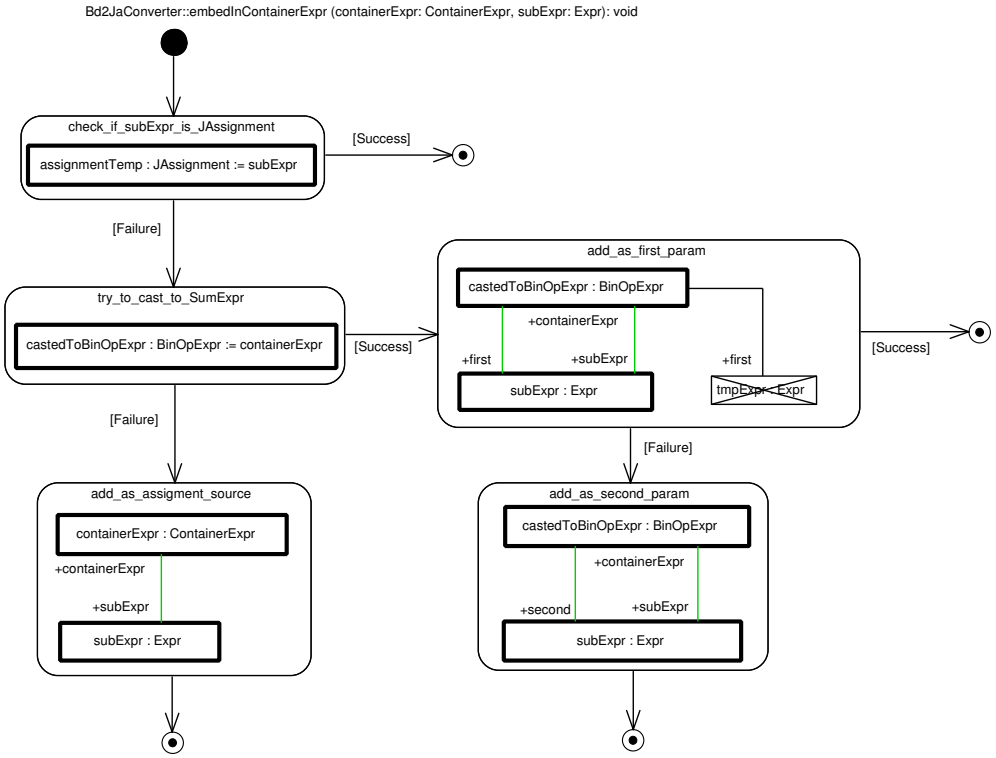


Abbildung A.13: `Bd2JaConverter::embedInContainerExpr(ContainerExpr, Expr)`

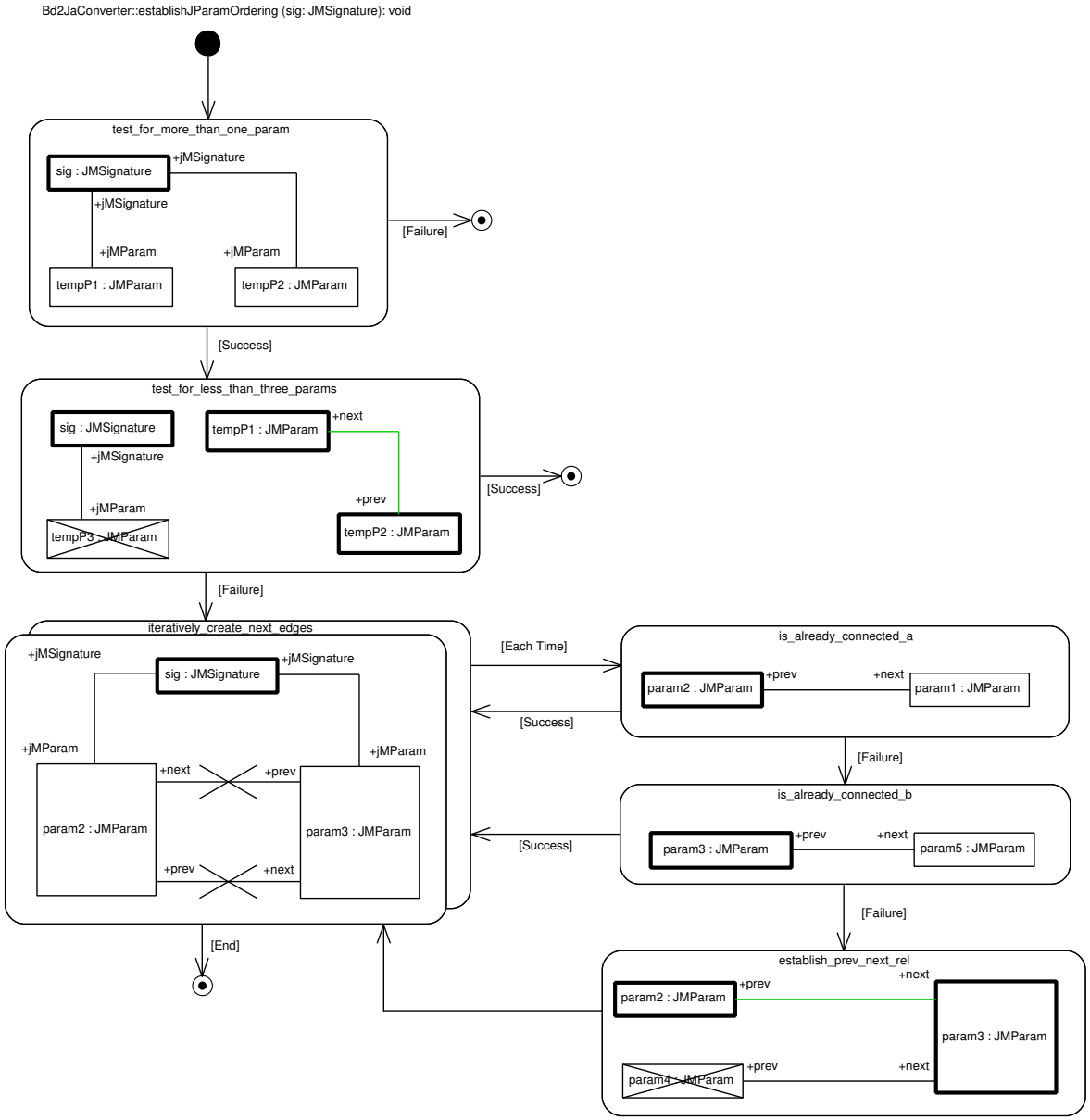


Abbildung A.14: `Bd2JaConverter::establishJParamOrdering(JMSignature)`

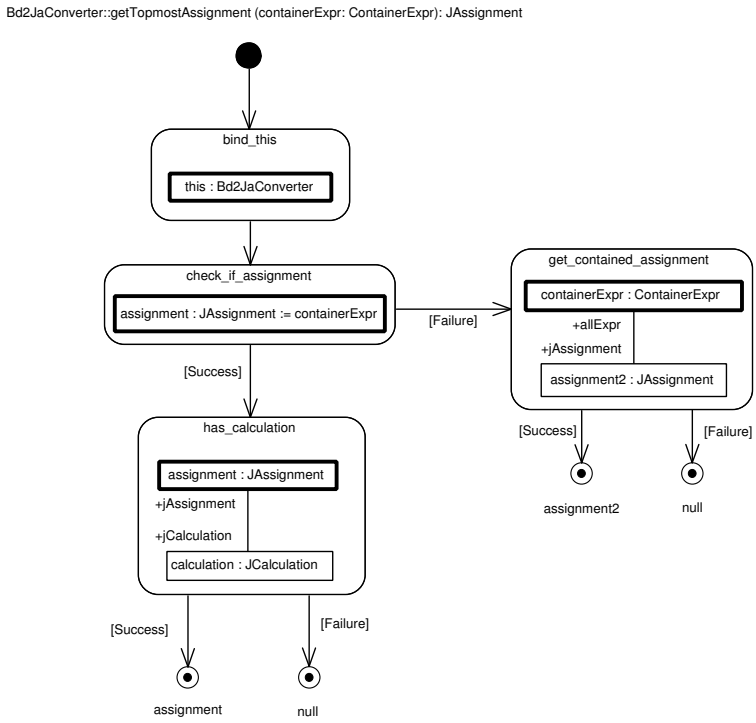


Abbildung A.15: `Bd2JaConverter::getTopmostAssignment(ContainerExpr)`

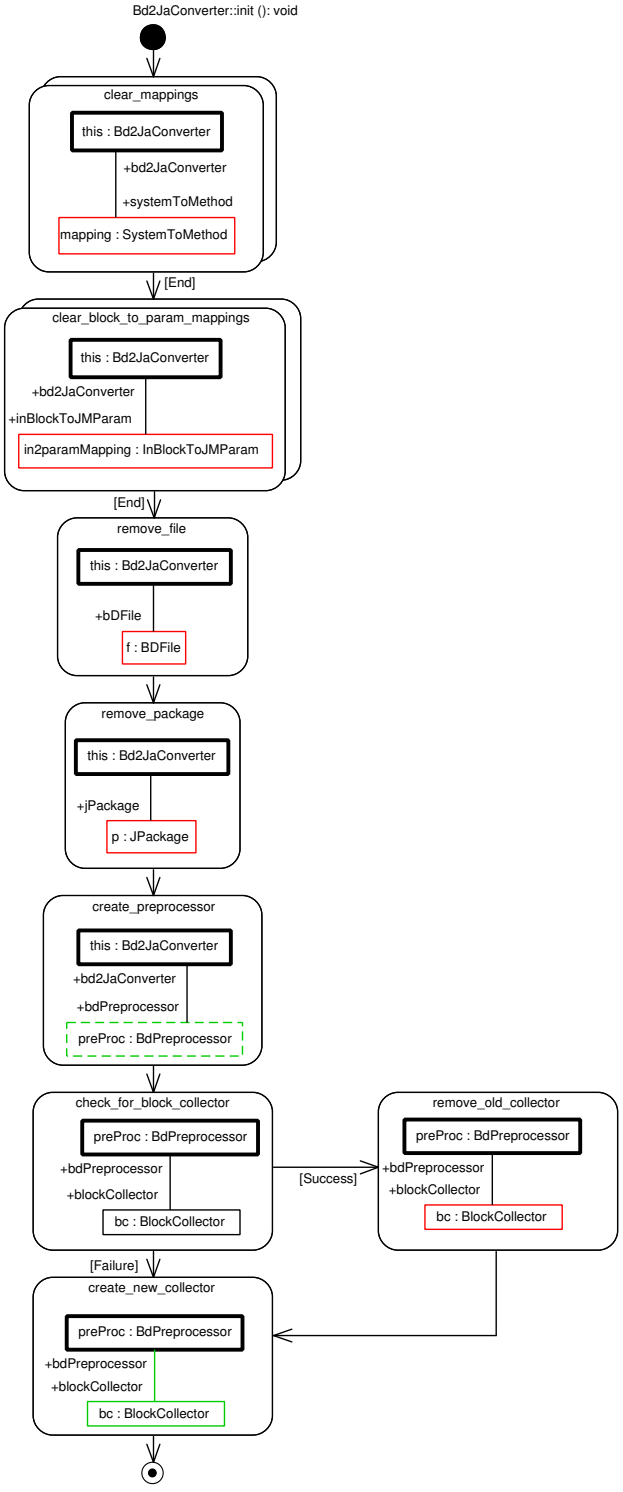


Abbildung A.16: `Bd2JaConverter::init()`

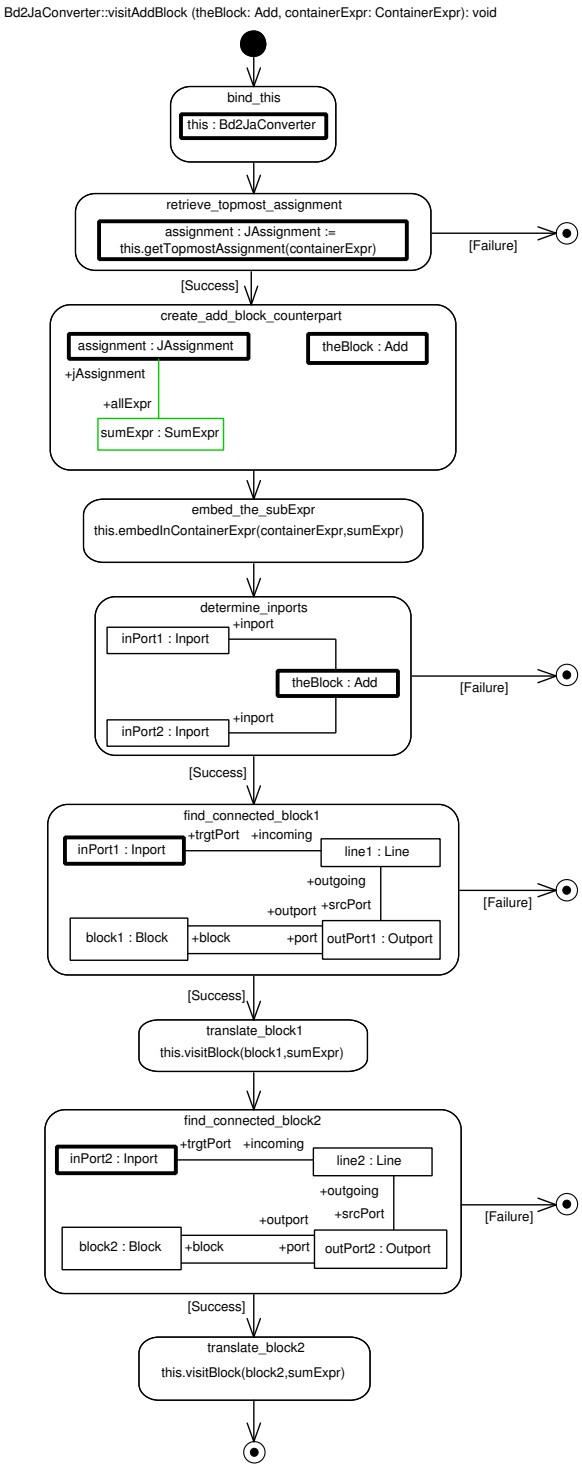


Abbildung A.17: `Bd2JaConverter::visitAddBlock(Add, ContainerExpr)`

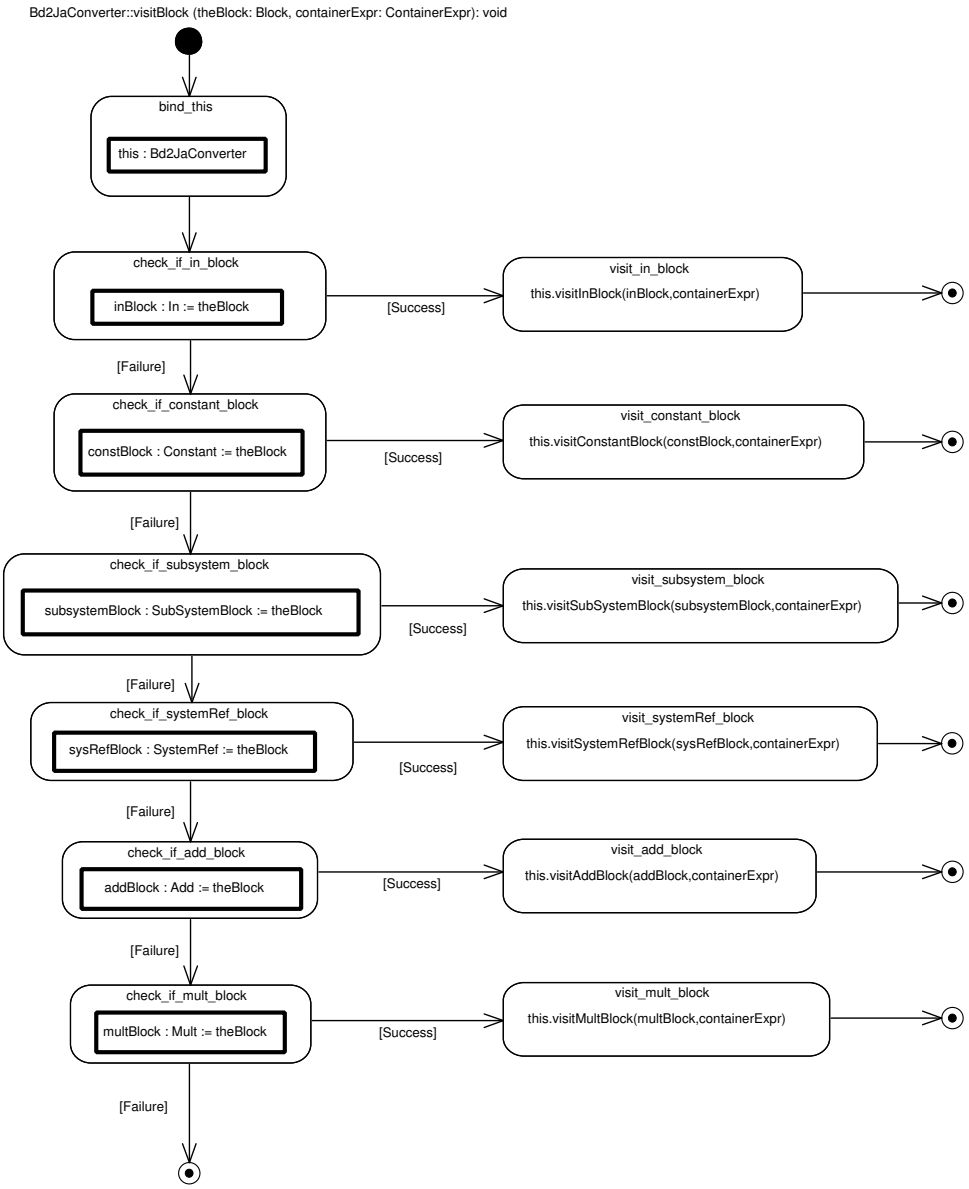


Abbildung A.18: `Bd2JaConverter::visitBlock(Block, ContainerExpr)`

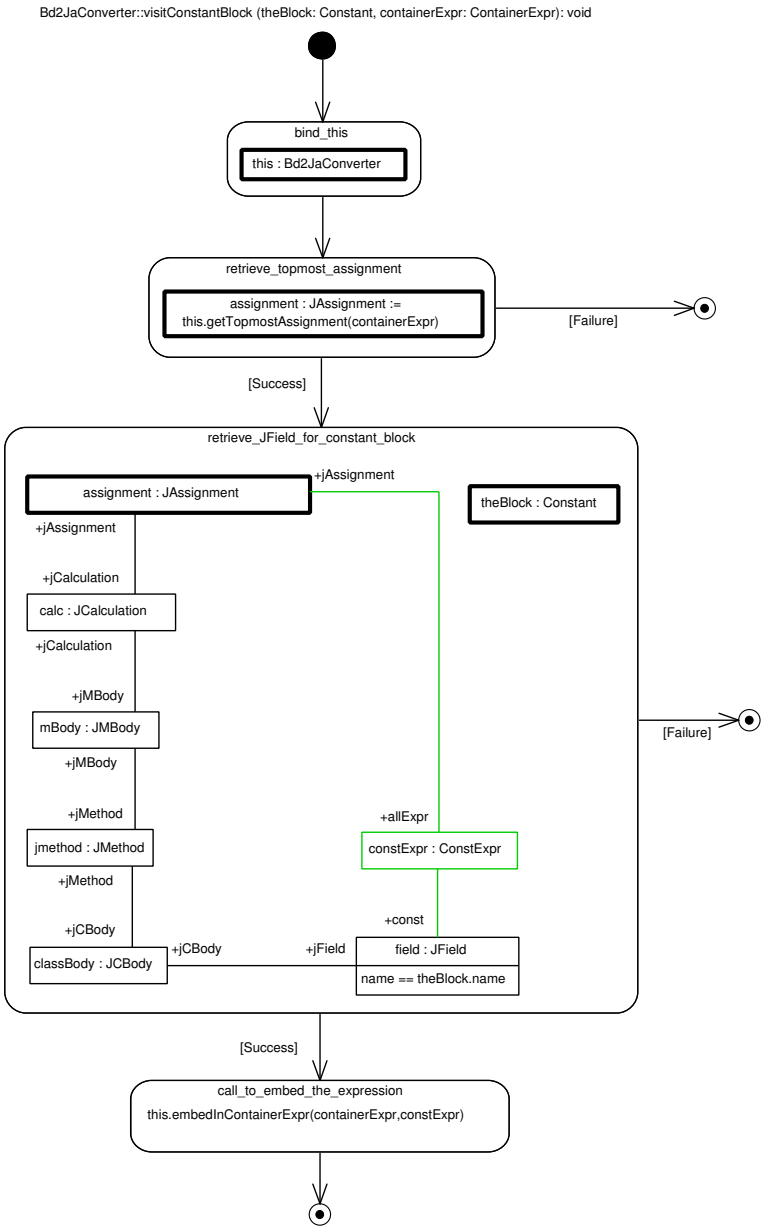


Abbildung A.19: `Bd2JaConverter::visitConstantBlock(Constant, ContainerExpr)`

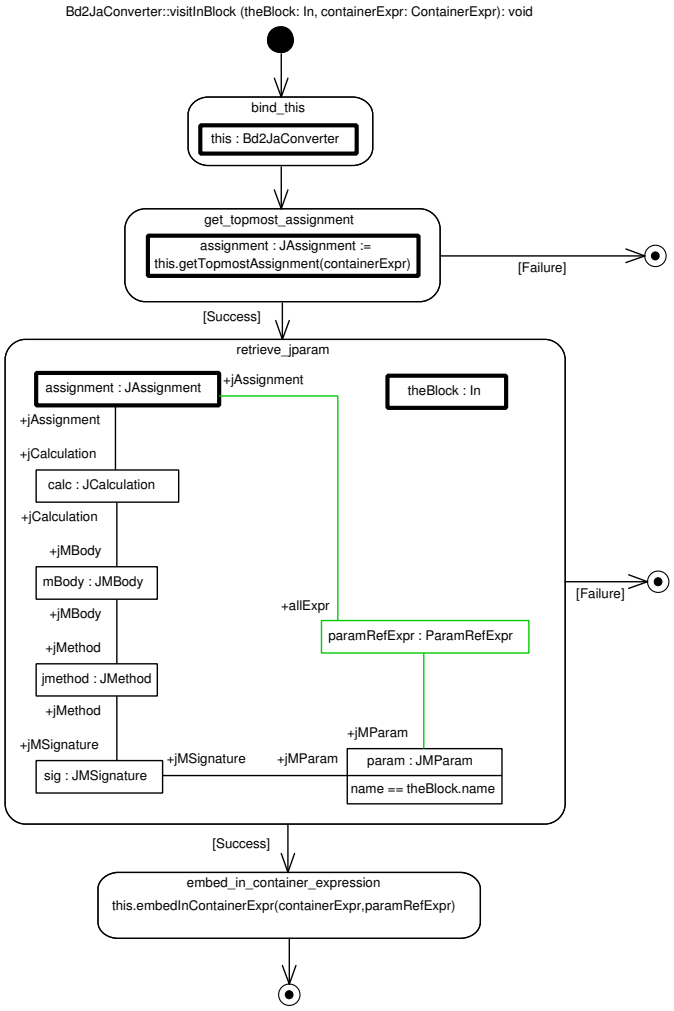


Abbildung A.20: `Bd2JaConverter::visitInBlock(In, ContainerExpr)`

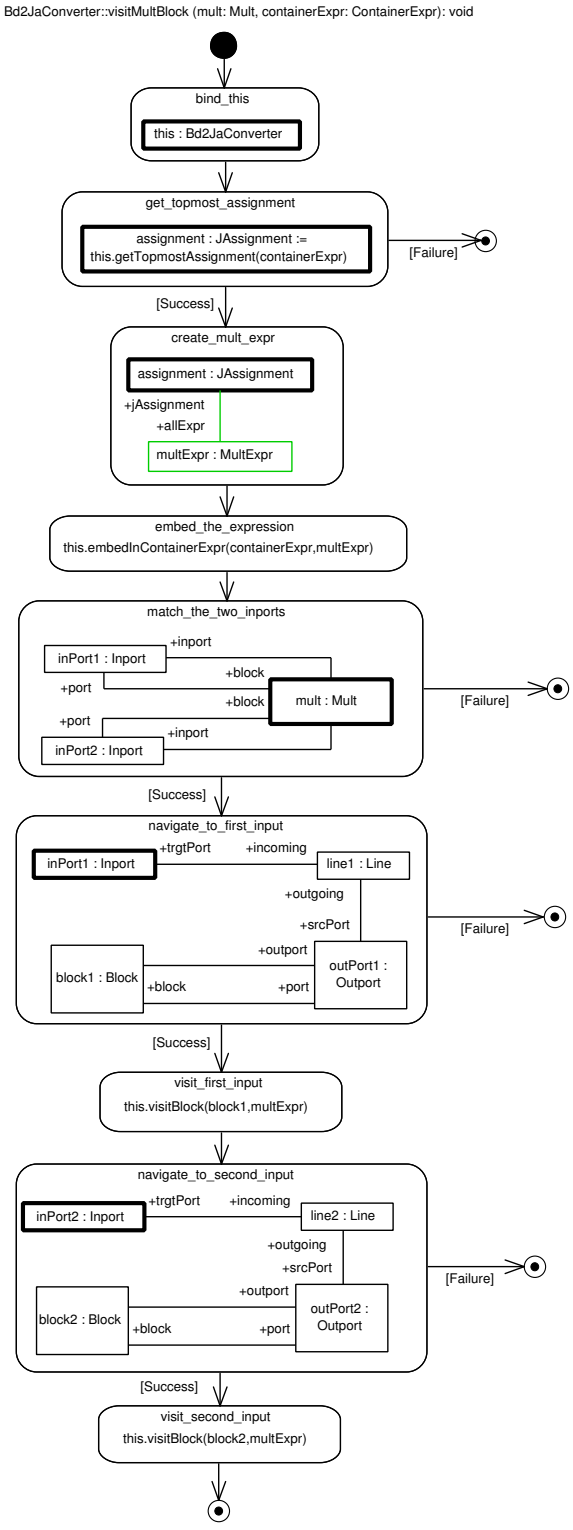


Abbildung A.21: `Bd2JaConverter::visitMultBlock(Mult, ContainerExpr)`

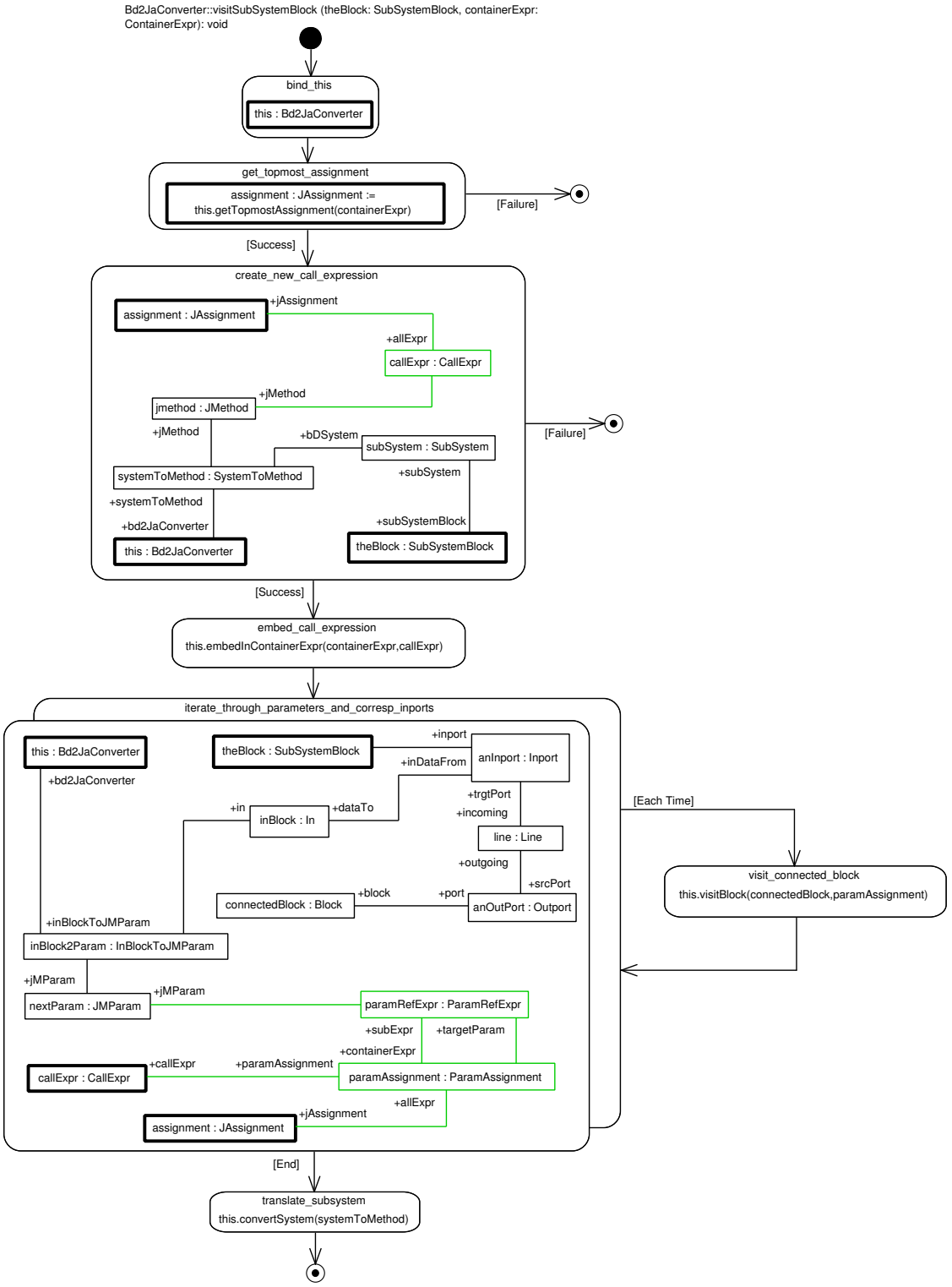


Abbildung A.22: `Bd2JaConverter::visitSubSystemBlock(SubSystemBlock, ContainerExpr)`

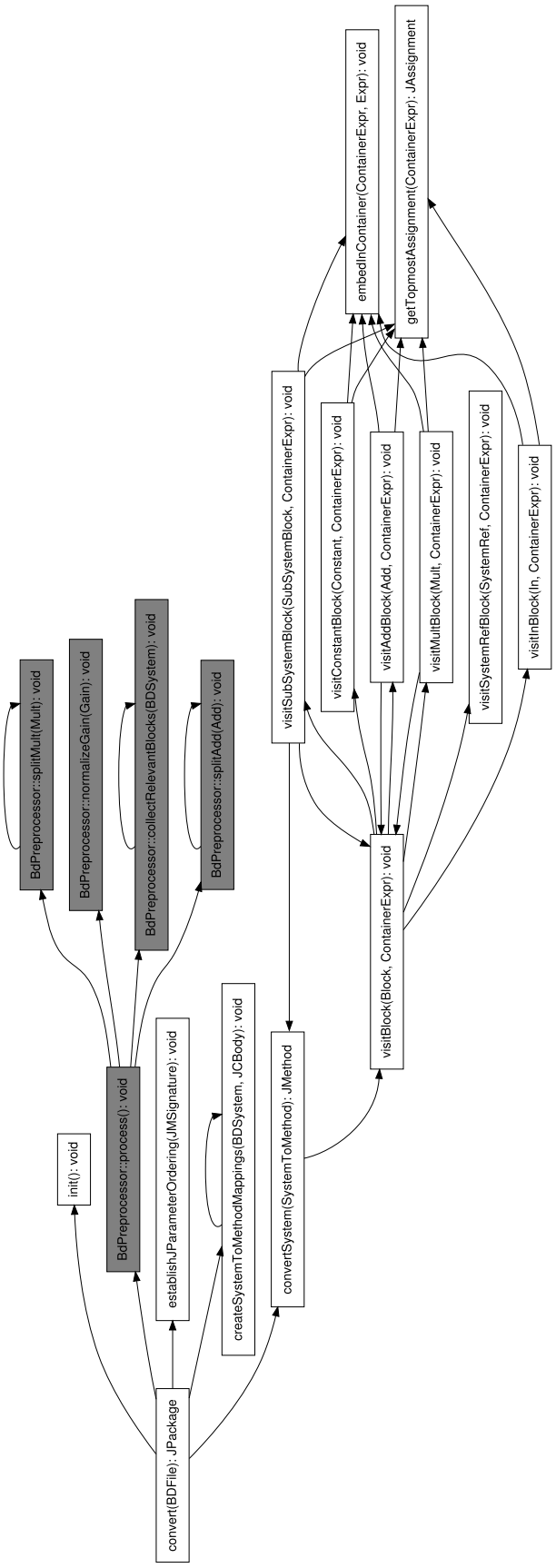


Abbildung A.23: Der Call-Graph der Bd2Ja-Transformation

B Zur LSCToMPN-Beispieltransformation

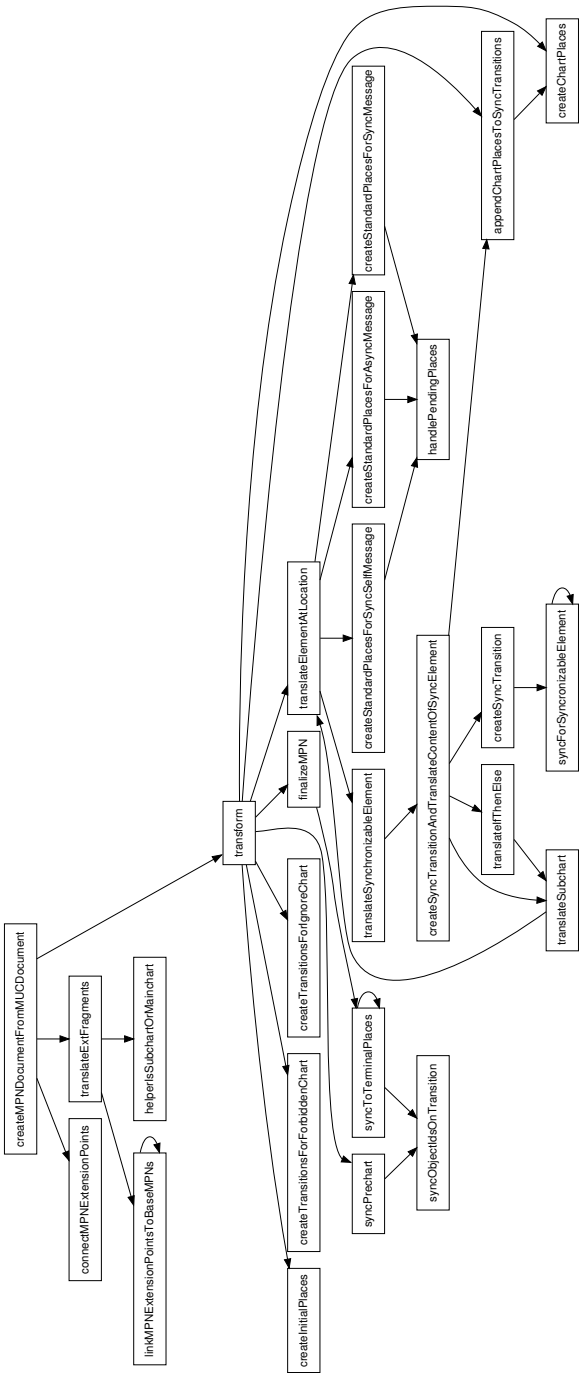
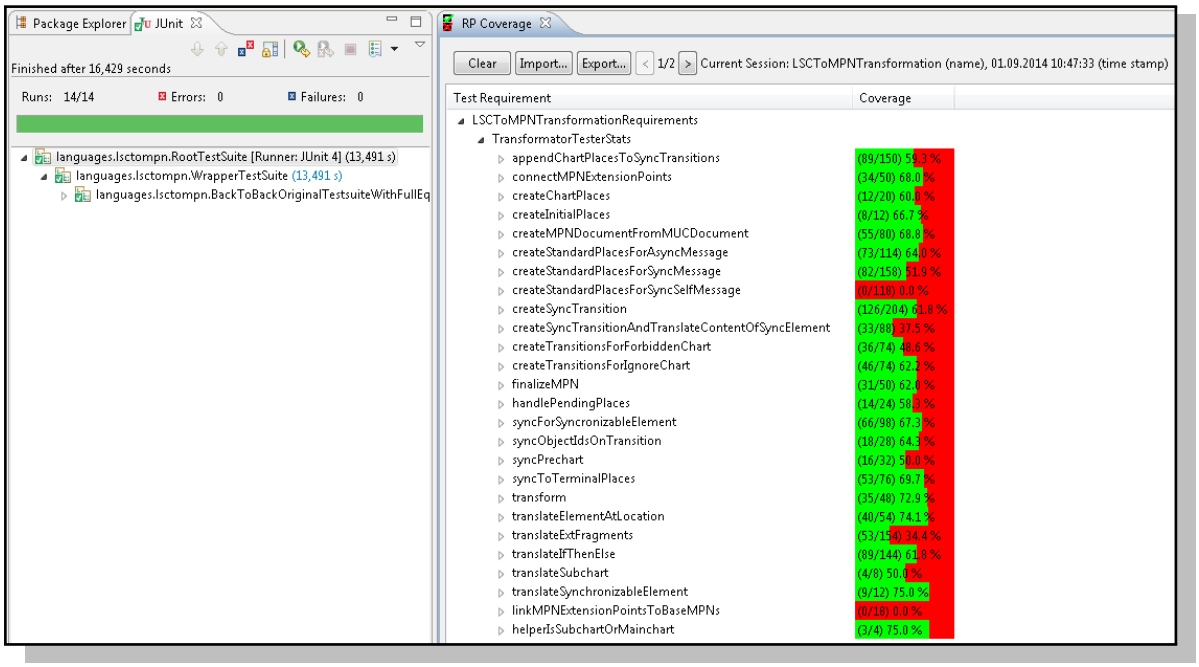
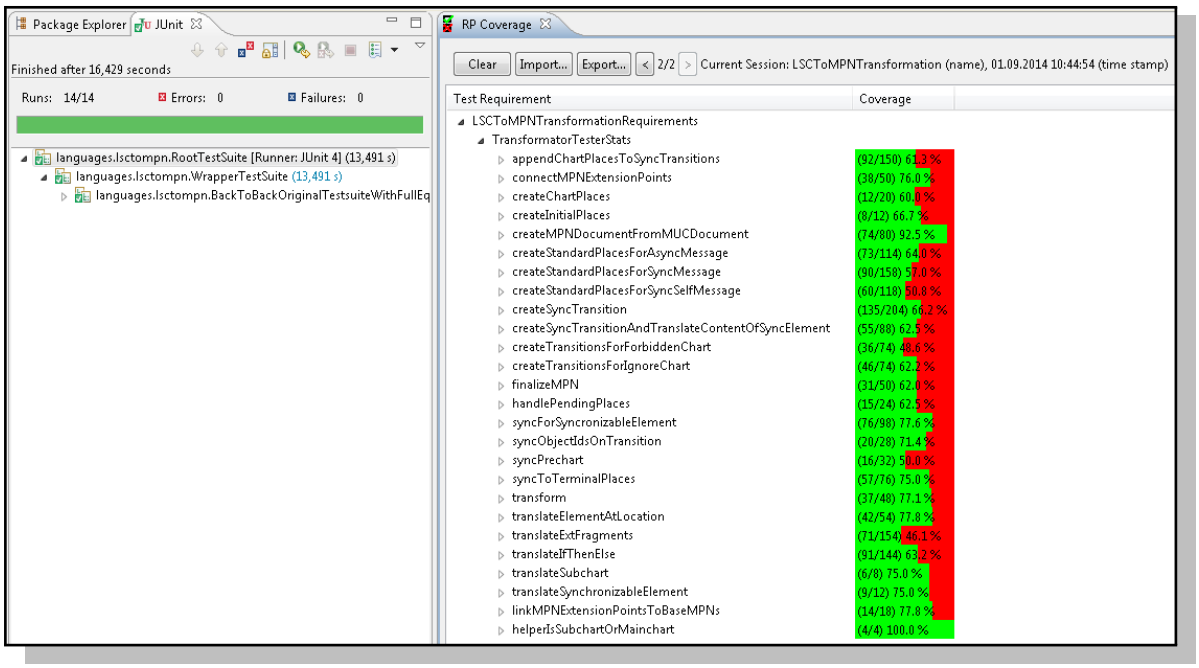


Abbildung B.1: Der Call-Graph der *LSCToMPN*-Transformation



(a) Überdeckung bei initialer Testmenge



(b) Überdeckung bei verbesserte Testmenge

Abbildung B.2: Visualisierung der *RP*-Überdeckungswerte für *LSCToMPN*

C Java-Code

C.1 Code der generierten *RPC*-Test-Suite

Listing C.1: Generierte Root-Test-Suite - installiert den `CoverageReportingSuiteRunner`

```
1  package theTestsuite;
2
3  /* omitted junit related imports */
4  import org.moflon.sdmtestcoverageframework.testsuite.CoverageReporting ↩
5                SuiteRunner;
6
7  // Use our special RP-Coverage TestSuiteRunner
8  @RunWith( CoverageReportingSuiteRunner.class )
9  @SuiteClasses({ CustomizableTestSuite.class }) // wrapped test suites (and ↩
       tests)
10 public class RootTestSuite {
11    @BeforeClass
12    public static void setUpClass() {
13       System.out.println("Master TS setup");
14    }
15
16    @AfterClass
17    public static void tearDownClass() {
18       System.out.println("Master TS tearDown");
19    }
20 }
```

Listing C.2: Durch den Benutzer konfigurierbare Test-Suite

```
1  package theTestsuite;
2
3  /* omitted junit related imports */
4  import BD2Ja.Bd2JaConverterBasicTests;
5  import BD2Ja.BdPreprocessorTests;
6
7  @RunWith(value = Suite.class)
8  @SuiteClasses(value = {
9         // configure your tests below
10        Bd2JaConverterBasicTests.class,
11        BdPreprocessorTests.class })
12 public class CustomizableTestSuite { /* intentionally left blank */ }
```

C.2 Code der *RPC*-Integration für JUnit

Listing C.3: Angepasster *RPC*-Test-Runner

```
1  package org.moflon.sdmtestcoverageframework.testsuite;
2
3  import java.util.*;
4  import org.junit.rules.TestRule;
5  import org.junit.runner.Description;
6  import org.junit.runners.BlockJUnit4ClassRunner;
7  import org.junit.runners.model.FrameworkMethod;
8  import org.junit.runners.model.InitializationError;
9  import org.junit.runners.model.Statement;
10 import org.moflon.sdmtestcoverageframework.protocol.messages.EndTestMessage;
11 import org.moflon.sdmtestcoverageframework.protocol.messages.NewTestMessage;
12
13 /*
14  * This class serves two purposes:
15  *   1) It sorts test cases so that a stable ordering is established
16  *      (required to deterministically obtain the indices of first coverage
17  *      occurrences)
18  *   2) It returns a custom TestRule implementation that takes care of view
19  *      notification.
20  */
21 public class CoverageReportingTestRunner extends BlockJUnit4ClassRunner {
22
23    private final TestRule myTestRule = new ReportingTestRule();
24
25    public CoverageReportingTestRunner(Class<?> clazz) throws ↩
          InitializationError {
26       super(clazz);
27    }
28
29    @Override
30    protected List<FrameworkMethod> computeTestMethods() {
31       List<FrameworkMethod> sortedMethods = new ↩
             ArrayList<FrameworkMethod>(super.computeTestMethods());
32       Collections.sort(sortedMethods, new Comparator<FrameworkMethod>() {
33          public int compare(FrameworkMethod method1, FrameworkMethod ↩
                method2) {
34             return method1.getName().compareTo(method2.getName());
35          }
36       });
37       return sortedMethods;
38    }
39
40    @Override
41    protected List<TestRule> getTestRules(Object target) {
42       List<TestRule> testRules = super.getTestRules(target);
43       if (!testRules.contains(myTestRule))
```

```
44          testRules.add(myTestRule);
45       return testRules;
46    }
47
48    /*
49     * This TestRule implementation takes care of informing the view (via
50     * the CoverageReportingChannel)
51     */
52    class ReportingTestRule implements TestRule {
53       @Override
54       public Statement apply(final Statement base, final Description ↩
             description) {
55          Statement statement = new Statement() {
56             @Override
57             public void evaluate() throws Throwable {
58                CoverageReportingChannel.getInstance().sendMsg(new ↩
                      NewTestMessage(description.getMethodName()));
59                try {
60                   base.evaluate();
61                } catch (Throwable t) {
62                   throw t;
63                } finally {
64                   CoverageReportingChannel.getInstance().tick();
65                   CoverageReportingChannel.getInstance().sendMsg(new ↩
                         EndTestMessage());
66                }
67             }
68          };
69          return statement;
70       }
71    }
72 }
```

Listing C.4: Kommunikationskanal - wichtige Methoden

```
1  package org.moflon.sdmtestcoverageframework.testsuite;
2
3  import org.moflon.sdmtestcoverageframework.protocol.messages.AbstractRp ↩
4                CoverageMessage;
5
6  abstract class BasicCoverageReportingChannel {
7         protected abstract void setup();
8         protected abstract void sendMsg(AbstractRpCoverageMessage msg);
9         protected abstract void tick();
10        protected abstract void shutdown();
11 }
```

Listing C.5: Implementierung des Kommunikationskanals

```
1  package org.moflon.sdmtestcoverageframework.testsuite;
2
3  /* imports ...*/
4
5  class CoverageReportingChannel extends BasicCoverageReportingChannel {
6     public static final int DEFAULT_PORT = 8137;
7     private static CoverageReportingChannel instance; // Singleton pattern
8
9     public static CoverageReportingChannel getInstance() {
10       if (instance == null) {
11          instance = new CoverageReportingChannel();
12       }
13       return instance;
14    }
15
16    private Socket sock; // communication channel
17    private BufferedReader in;
18    private PrintWriter out;
19
20    private CoverageReportingChannel() {
21       setup();
22    }
23
24    @Override
25    protected void setup() {
26       try {
27          // setup communication channel
28          sock = new Socket("localhost", DEFAULT_PORT);
29          out = new PrintWriter(sock.getOutputStream(), true);
30          in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
31
32          // send first message
33          setupTheSocketConnection();
34          String projectName = getProjectName();
35          sendMsg(new NewSessionMessage(projectName));
36          ...
37       } catch (.) { ... }
38    }
39
40    @Override
41    protected void sendMsg(AbstractRpCoverageMessage msg) {
42       try {
43         out.println(msg.toString());
44         ...
45       } catch (IOException e) { ... }
46    }
47
48    @Override
```

```
49    protected void tick() {
50       ...
51       // send update messages for the collected coverage data
52       sendMsg(new NewCovItemMessage(payload));
53       ...
54       sendMsg(new TickMessage());
55    }
56
57    @Override
58    protected void shutdown() {
59       sendMsg(new EndSessionMessage());
60       out.close();
61    }
62
63    ...
64 }
```

D Das EMF-Ecore-Metamodell

Der Abschnitt zeigt das Ecore-Metamodell in einer inoffiziellen Darstellung. Vgl. hierzu auch [Ste+09] bzw. `http://eclipse.org/modeling/emf/` (Stand: 6.1.2015).

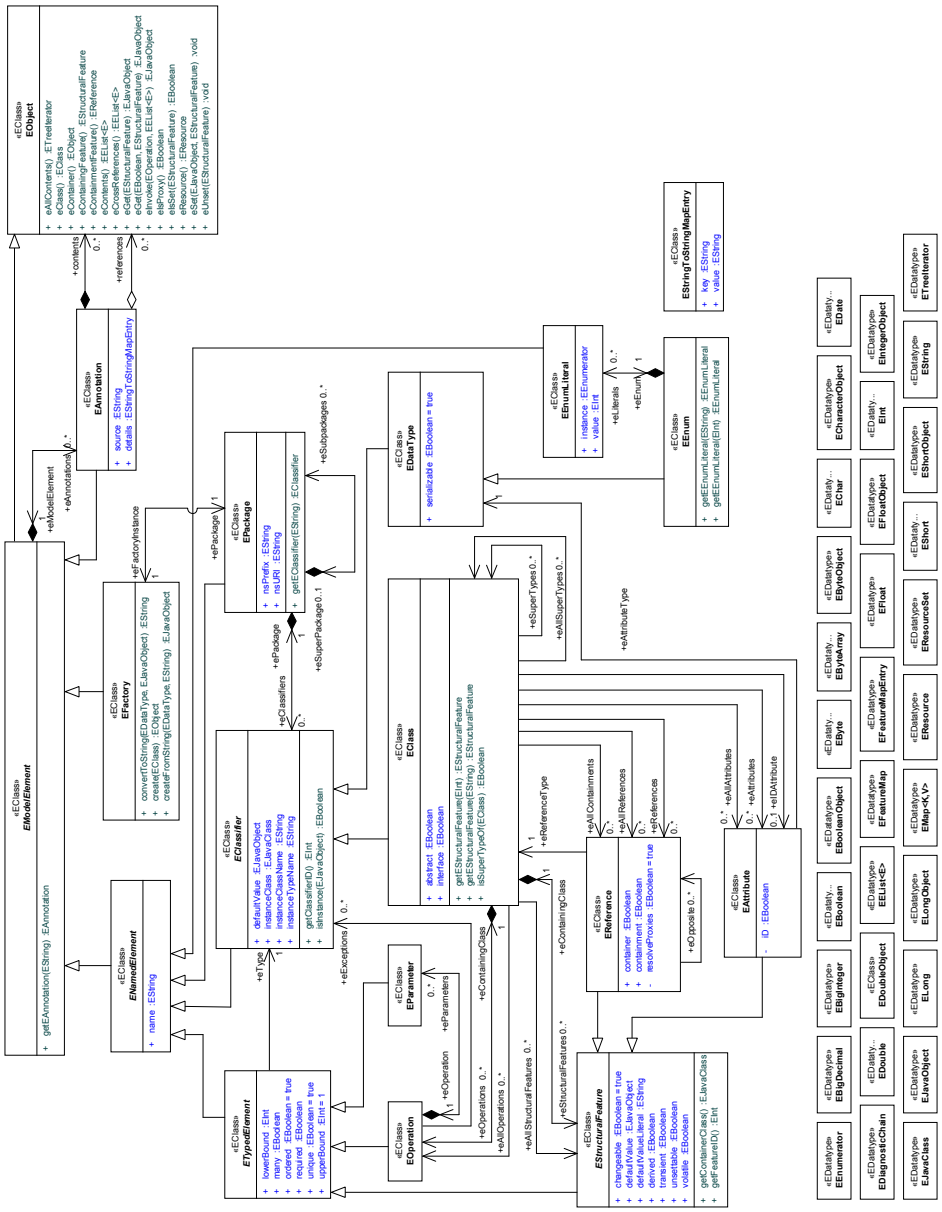


Abbildung D.1: Die eMoflon-Variante des EMF-Ecore-Metamodells

E Das SDM-Metamodell

Dieser Abschnitt enthält Klassendiagrammdarstellungen für die Pakete des *SDM*-Metamodells von eMoflon[1] inkl. einiger Referenzen zu EMF/Ecore. Das Metamodell ist das Hauptergebnis des *sdm-commons*-Projektes,[2] welches als Ziel die Entwicklung eines werkzeugübergreifenden Metamodells für Story-Diagramme hatte.

Das Layout der Diagramme basiert auf der Enterprise-Architect-Variante aus dem eMoflon-Projekt (Stand: Januar 2014). Das Diagrammlayout wurden teilweise für die Darstellung in dieser Arbeit angepasst und optimiert. Die Abbildung E.8 ist selbst erstellt. Sie fasst Konzepte für *SDM*-Expressions aus unterschiedlichen Paketen als Sicht zusammen und nicht Teil des ursprünglichen Metamodells.

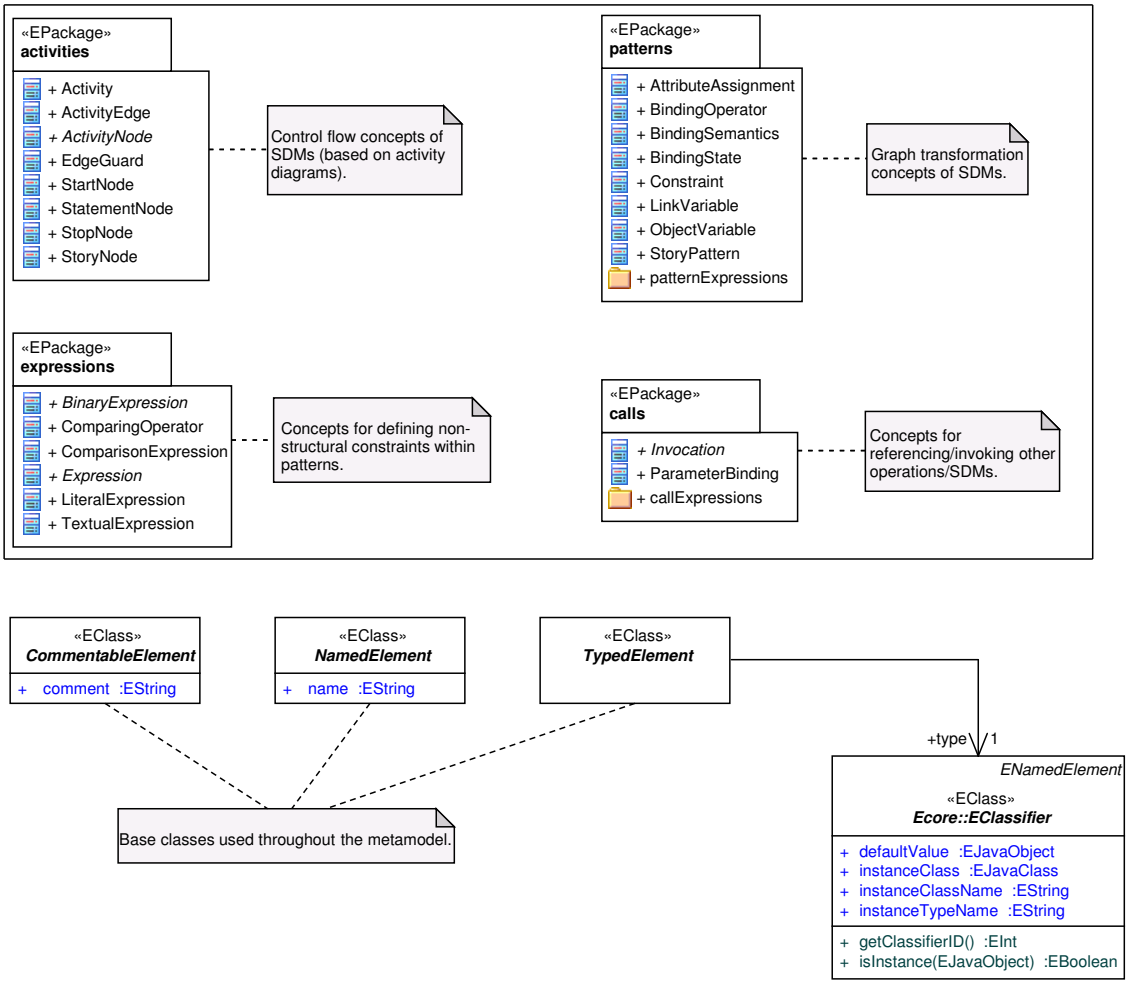


Abbildung E.1: Das Basispaket des SDM-Metamodells

[1] `http://www.emoflon.org`

[2] `https://code.google.com/a/eclipselabs.org/p/sdm-commons/` (zuletzt abgerufen am)

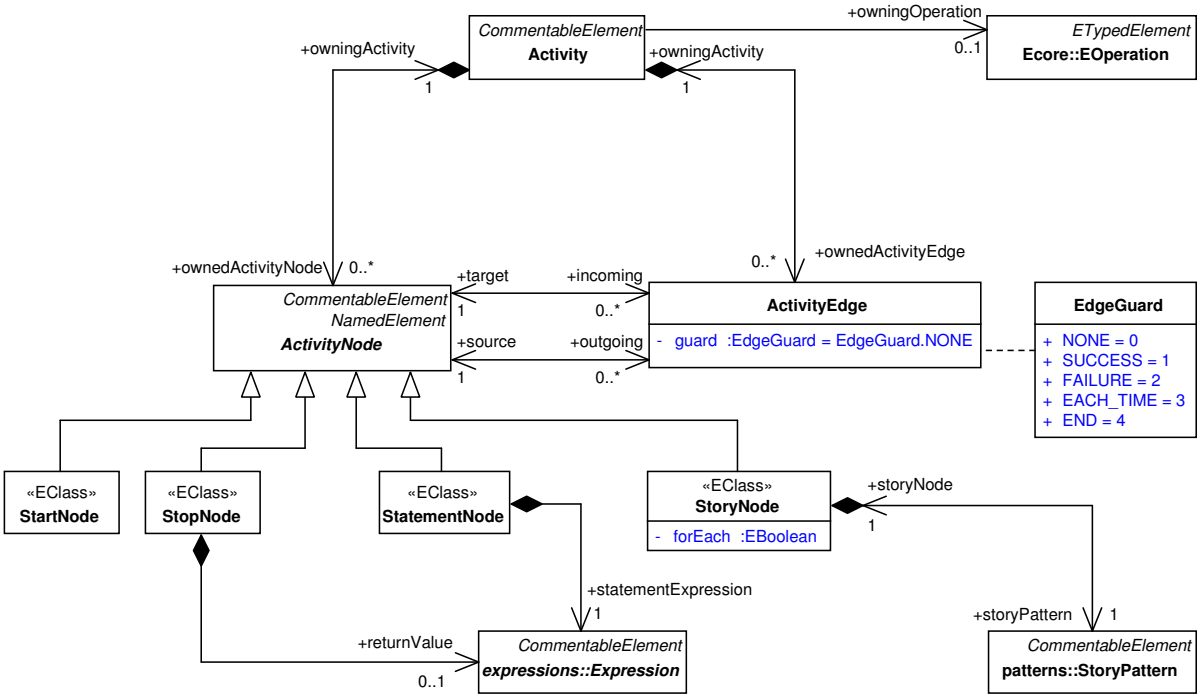


Abbildung E.2: Das `activities`-Paket

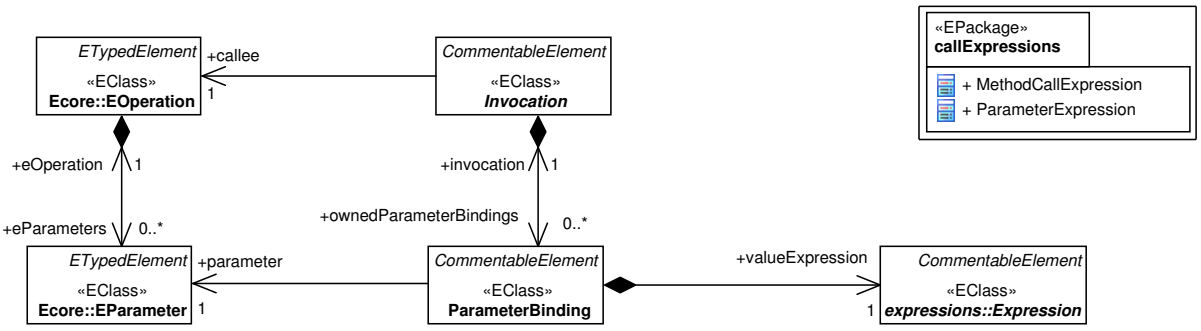


Abbildung E.3: Das `calls`-Paket

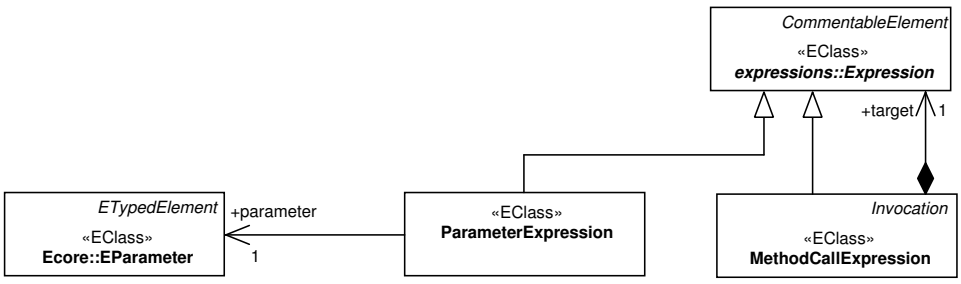


Abbildung E.4: Das `callExpressions`-Unterpaket von `calls`

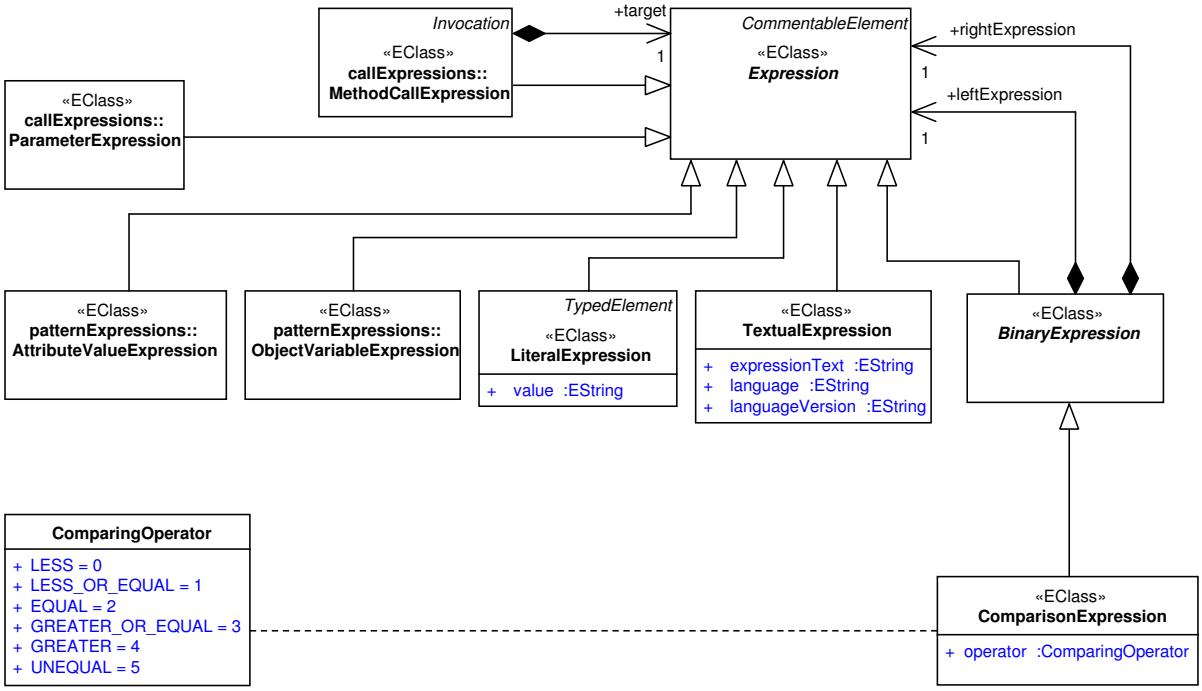


Abbildung E.5: Das `expressions`-Paket

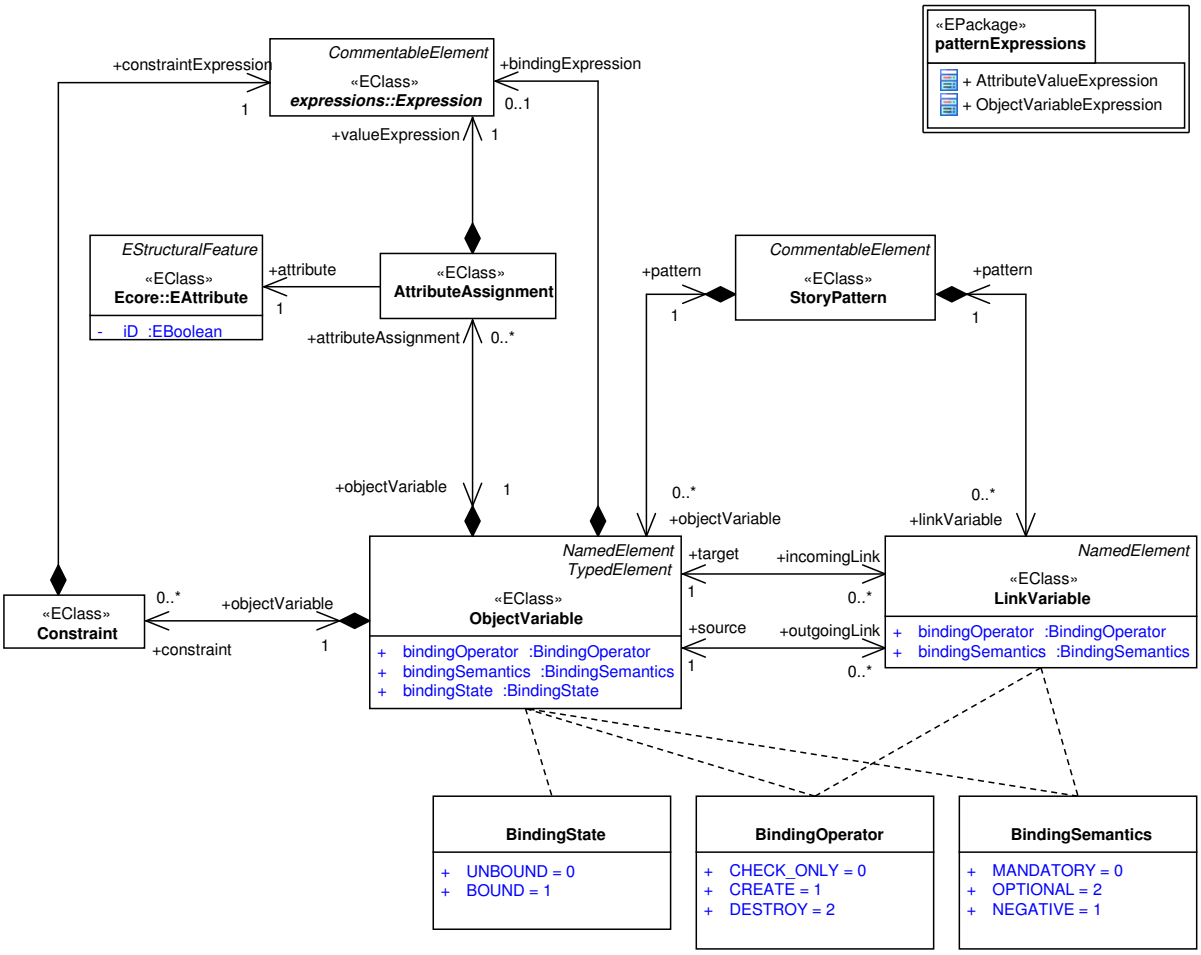


Abbildung E.6: Das `patterns`-Paket

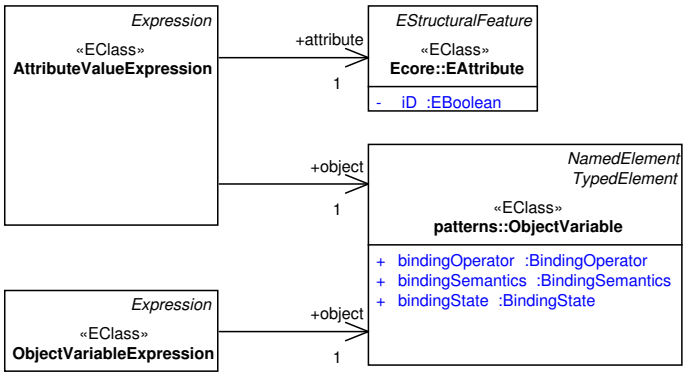


Abbildung E.7: Das `patternExpressions`-Unterpaket von `patterns`

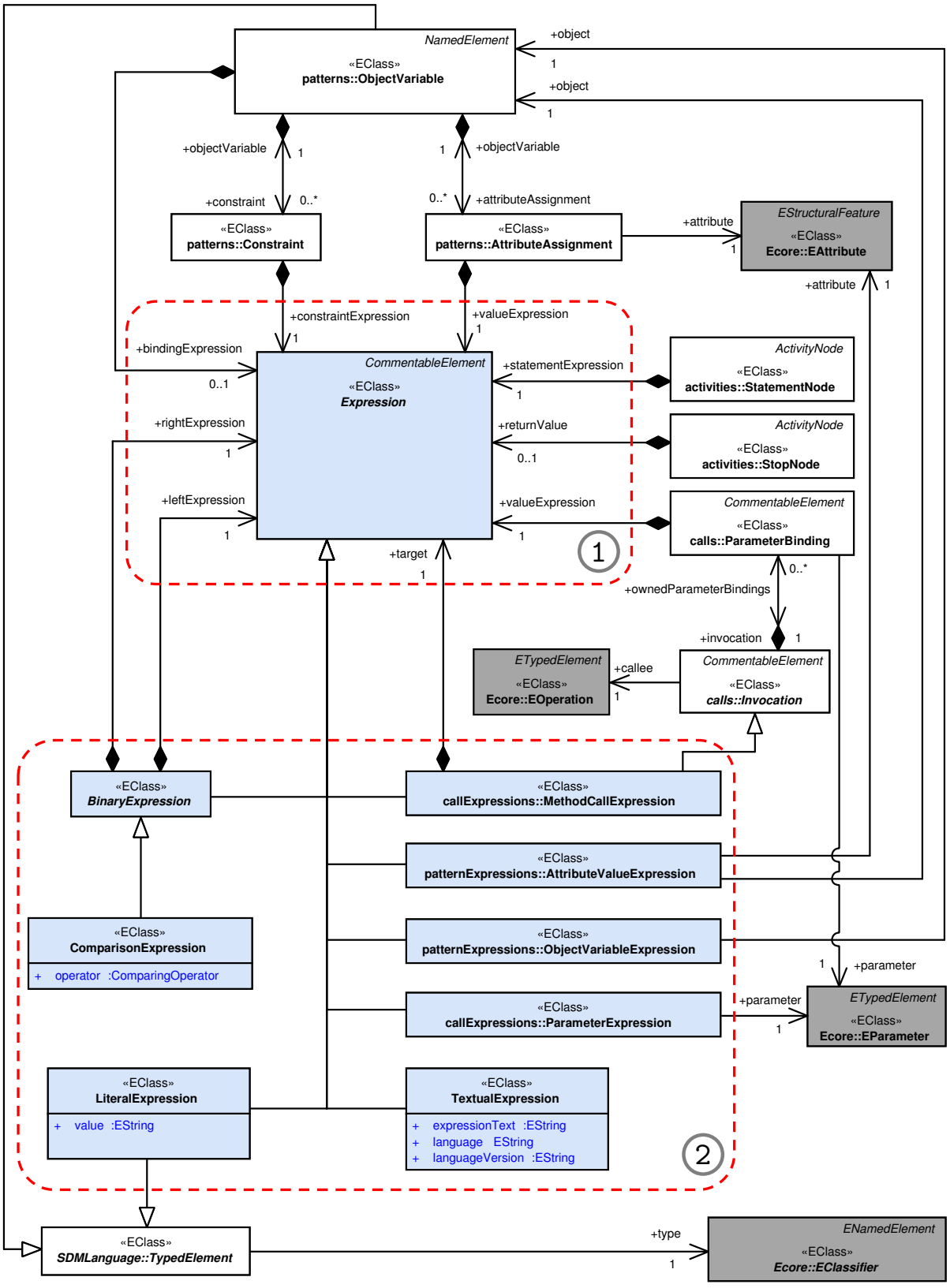


Abbildung E.8: Verwendung und Unterklassen von *SDM*-Expression

F Syntax der textuellen Anteile der SDM-Sprache

```
<ObjectVariableLabel>    = 'this' | <OvName> ':' <OvType> [ ':=' <Expr> ] ;
<StatementNodeLabel>     = <OpCallExpr> ;
<StopNodeLabel>          = <Expr> ;
<AttributeAssignmentLabel> = <AttribValueExpr> ':=' <Expr> ;
<AttributeConstraintLabel> = <AttribValueExpr> <ComparisonOperator> <Expr> ;
<ComparisonOperator>     = '==' | '!=' | '>=' | '<=' | '>' | '<' ;
```

```
<Expr> = <OpCallExpr> | <AttribValueExpr> | <ParamRefExpr> | <OvExpr> | ↩
   <LiteralExpr> ;

<OpCallExpr> = ( <OvExpr> | <ParamRefExpr> ) '.' <OpName> '(' ↩
   <ParamValueExpr> ')' ;
<ParamValueExpr> = '' | <Expr> { ',' <Expr> } ;

<AttribValueExpr> = <OvExpr> '.' <AttrName> ;

<ParamRefExpr> = <ParamName> ;

<OvExpr> = <OvName> ;

<LiteralExpr> = ? java.lang.String-Literal ? ;

<OpName>   = ? JavaMethodName ? ;
<OvName>   = ? JavaIdentifier ? ;
<AttrName> = ? JavaIdentifier ? ;
<ParamName> = ? JavaIdentifier ? ;
```

Abkürzungen

API	Application Programming Interface. 21, 35, 54, 102, 238
ASM	Abstract State Machine [BS03]. 51
AST	Abstract Syntax Tree. 74
ATL	Atlas Transformation Language [Jou+06]. 119, 128, 129, 138, 140, 204, 277
BPEL	Business Process Execution Language. 120
BPMN	Business Process Model and Notation. 17, 136
CASE	Computer-Aided Software Engineering [Fug93]. 53
CFG	Control Flow Graph. 205–207, 217
CLP	Constraint Logic Programming. 112
CMOF	Complex MOF. 21
CPN	Colored Petri Net. 119
CSP	Constraint Satisfaction Problem. 68, 112
Diff	Differencing. 89
DPO	Double Pushout. 44, 46, 121
DSL	Domain Specific Language. 116
EBNF	Extended Backus Naur Form. 21, 97
EDV	Elektronische Datenverarbeitung. 36
eLSC	Extended *Live Sequence Chart (LSC)*. 247, 250
EMF	Eclipse Modeling Framework. 22, 23, 26, 37, 48, 49, 55, 68, 69, 109, 113, 201, 238–240, 282, 285
EMOF	Essential MOF. 21, 22
ER	Entity-Relationship. 17–19, 112
FSM	Finite State Machine. 93, 105, 106
fUML	Foundational UML [13a]. 110
GT	Graphtransformation. 4, 7, 41, 43–55, 57, 58, 61–64, 68, 72, 74, 75, 107, 117, 120–123, 126, 129, 133–141, 145, 146, 148, 149, 151, 154, 160, 161, 164, 166, 168, 170, 180, 187, 193, 194, 196, 201, 242, 250, 265, 275, 278, 279, 281, 283, 287
GUI	Graphical User Interface. 110, 182–184, 193
HAZOP	**Ha**zard and **Op**erability Studies [IEC01]. 203
HDL	Hardware Description Language. 106
HiL	Hardware in the Loop. 106

HOT	Higher-Order Transformation. 31, 238, 241, 279–281
IDE	Integrated Development Environment. 22, 112, 178, 184
JaCoCo	Java Code Coverage Library [14a]. 190, 193
JDT	Java Development Tools. 240
JET	Java Emitter Template. 23
JMI	Java Metadata Interface. 21, 22
JML	Java Modeling Language [LBR99]. 106
JRE	Java Runtime Environment (Laufzeitumgebung von Java). 267
JVM	Java Virtual Machine. 184, 236
KI	Künstliche Intelligenz. 107
LHS	Left Hand Side. 39, 43, 44, 49, 54, 62–65, 126, 127, 134, 139, 148, 150, 151, 160–162, 167, 187, 188, 195, 208, 263, 279
LOC	Lines of Codes. 253, 254
LSC	Live Sequence Chart. 247, 250, 262, 333
LTL	Linear-Time Temporal Logic, vgl. [HR04]. 122
LV	Link-Variable. 49, 63, 68–71, 160, 163, 166, 169–172, 188, 212–215, 222, 229, 230, 247, 252, 279
M2M	Model-to-Model. 30, 33, 34, 36, 37, 41, 118, 133, 134
M2T	Model-to-Text. 30, 33, 36, 37, 117, 118, 120, 125, 133, 134
MAJA	**Ma**tlab (Simulink und Stateflow) **J**ava **A**dapter. 35
MBT	Model-Based Testing. 6, 102, 103, 105–107, 125, 202
MCDC	Modified Condition Decision Coverage (MCDC), vgl. [CM94]. 96
MDA	Model Driven Architecture. 15, 17, 29, 30
MDD	Model Driven Development. 15, 20
MDE	Model Driven Engineering. 2, 15, 20, 114
MDSD	Model Driven Software Development. 3, 15, 16, 23, 29, 36, 37, 85, 103, 105, 108, 201, 277
MM	Metamodell. 238, 265
MOF	Meta Object Facility. 18, 21–23, 37, 48, 108, 118, 119
MOFM2T	MOF Model to Text Transformation Language. 37
MOMoC	**MO**F Meta-**Mo**del **C**ompiler. 36
MPN	Monitor Petri Net. 247, 250, 251
MSC	Message Sequence Chart. 17, 106, 247
MT	Modelltransformation (MT). 17, 30–32, 34–37, 41, 43, 53, 90, 105, 113–120, 124–129, 131, 133–135, 137, 138, 140, 201, 203–205, 222, 241, 280, 281, 288
NAC	Negative Application Condition. 45, 50, 64–66, 71, 130, 152, 160, 163–165, 167–171, 224, 260
OCL	Object Constraint Language. 18, 29, 106, 108, 112, 113, 116, 118, 119, 121–128, 130, 205

OMG	Object Management Group. 15, 17, 18, 21, 29, 37, 107
OO	Object-Oriented (objektorientiert). 15, 17, 18, 53, 201, 202, 204, 211
OV	Object-Variable. 49, 62–73, 75, 76, 149, 152, 160–174, 188, 209, 211–215, 217, 221–229, 247, 252, 253, 263, 279, 283
PIM	Platform Independent Model. 29, 33
PSM	Platform Specific Model. 29, 33
QVT	Query/View/Transformation [OMG11]. 29, 35, 38, 45, 116, 119, 277
RHS	Right Hand Side. 39, 43, 44, 49, 54, 62–65, 167
RP	Requirement-Pattern. 151–155, 157, 158, 160–162, 164–175, 178, 180–191, 193–197, 245, 246, 250–254, 258, 260–266, 268, 269, 271, 272, 274–276, 279–285, 316
RPC	Requirement Pattern Coverage. 154, 178–182, 184, 185, 187, 188, 190, 191, 193–197, 210, 211, 222, 238, 242, 245, 246, 250–252, 254, 258, 260, 261, 266–276, 278–283, 285–287, 318
SAT	Boolean Satisfiability (Erfüllbarkeitsproblem der Aussagenlogik). 112, 113
SDK	Software Development Kit. 257
SDM	Story Driven Modeling. 9, 11, 12, 53–68, 71–74, 77, 107, 108, 110, 138, 139, 141, 145, 146, 148, 150–153, 155, 158, 160, 161, 169, 171, 172, 174–176, 180, 187, 188, 190, 191, 193–219, 221–223, 228, 230, 232, 233, 235, 238, 239, 241, 242, 245, 247, 248, 250–253, 255, 257, 258, 261, 265, 280–285, 287, 291, 325, 329
SMT	Satisfiability Modulo Theories. 118
SPO	Single Pushout. 46, 54, 121, 122
SQL	(oft) Structured Query Language. 35
SUT	System Under Test. 11, 80, 82, 83, 86–89, 92, 96, 98, 99, 101–103, 131, 139, 153, 155, 158, 164, 180, 182, 186, 200, 275, 284, 285, 288
TGG	Triple Graph Grammar. 120, 123, 131, 135–137, 287
UI	User Interface. 258
UML	Unified Modeling Language. 16–21, 23, 26, 53–56, 62, 94, 105–110, 112, 118, 125, 127, 134, 201–203
UTP	UML Testing Profile [13b]. 107
XMI	XML Metadata Interchange. 21–23, 35, 131, 211
XML	eXtensible Markup Language. 21, 35, 36, 90, 131, 211
XSLT	XSL Transformation. 35, 36

Literatur

[01] *Model Driven Architecture (MDA)*. http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf. Entwurf, Dokument „ormsc/2001-07-01“. Architecture Board ORMSC, Object Management Group, 2001 (siehe S. 15).

[02] *Java Metadata Interface (JMI) Specification*. JSR 040 http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html (zuletzt abgerufen am 23.10.2014). Juni 2002 (siehe S. 21).

[03a] *MDA Guide Version 1.0.1*. Dokument „omg/2003-06-01“. Object Management Group. 2003 (siehe S. 15, 29, 30).

[03b] *OMG Unified Modeling Language Specification, Version 1.5*. http://www.omg.org/spec/UML/1.5/. Dokument „formal/03-03-01“. Object Management Group, 2003 (siehe S. 57, 107).

[04] *Duden: Die deutsche Rechtschreibung*. 23. Aufl. Bd. 1. Der Duden in zwölf Bänden. Dudenredaktion, 2004 (siehe S. 29).

[08] *MOF Model to Text Transformation Language, v1.0*. http://www.omg.org/spec/MOFM2T/1.0/PDF. Dokument „formal/2008-01-16“. Object Management Group, Jan. 2008 (siehe S. 37).

[09] „IEEE Standard VHDL Language Reference Manual“. In: *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan. 2009), S. c1–626. DOI: 10.1109/IEEESTD.2009.4772740.

[11a] *OMG Meta Object Facility (MOF) Core Specification*. http://www.omg.org/spec/MOF/2.4.1/PDF/. Dokument „formal/2011-08-07“. Object Management Group, 2011 (siehe S. 20, 21).

[11b] *OMG Unified Modeling Language (OMG UML), Infrastructure*. http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/. Dokument „formal/2011-08-05“. Object Management Group, Aug. 2011 (siehe S. 17, 18, 21, 22).

[11c] *OMG Unified Modeling Language (OMG UML), Superstructure*. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/. Dokument „formal/2011-08-05“. Object Management Group, Aug. 2011 (siehe S. 17–19, 23, 55, 112).

[12] *V-Modell XT*. http://www.v-modell-xt.de/. Die Beauftragte der Bundesregierung für Informationstechnik (CIO Bund), Bundesministerium des Innern, Abteilung IT, 2012 (siehe S. 2).

[13a] *Semantics of a Foundational Subset for Executable UML Models (fUML)*. http://www.omg.org/spec/FUML/1.1/PDF/. Dokument „formal/2013-08-06“. Object Management Group, Aug. 2013 (siehe S. 110, 333).

[13b] *UML Testing Profile (UTP)*. http://www.omg.org/spec/UTP/1.2/PDF/. Dokument „formal/2013-04-03“. Object Management Group, Apr. 2013 (siehe S. 107, 335).

[14a] *JaCoCo Java Code Coverage Library*. http://www.eclemma.org/jacoco/. (zuletzt abgerufen am 30.3.2015). 2014 (siehe S. 334).

[14b] *OMG XML Metadata Interchange (XMI) Specification*. http://www.omg.org/spec/MOF/2.4.1/PDF/. Dokument „formal/2014-04-04“. Object Management Group, 2014 (siehe S. 21).

[AB11] M. F. van Amstel und M. G. J. van den Brand. „Model Transformation Analysis: Staying Ahead of the Maintenance Nightmare“. In: *Theory and Practice of Model Transformations*. Hrsg. von J. Cabot und E. Visser. Bd. 6707. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 108–122. ISBN: 978-3-642-21731-9. DOI: 10.1007/978-3-642-21732-6_8 (siehe S. 3, 114).

[ABK07] K. Anastasakis, B. Bordbar und J. M. Küster. „Analysis of Model Transformations via Alloy“. In: *Proceedings of the 4th MoDeVVa workshop Model-Driven Engineering, Verification and Validation*. Hrsg. von B. Baudry, A. Faivre, S. Ghosh und A. Pretschner. 2007, S. 47–56 (siehe S. 118, 122).

[ABM98] P. Ammann, P. Black und W. Majurski. „Using model checking to generate tests from specifications“. In: *Formal Engineering Methods, 1998. Proceedings. Second International Conference on*. Dez. 1998, S. 46–54. DOI: 10.1109/ICFEM.1998.730569 (siehe S. 106).

[Agr+05] A. Agrawal, A. Vizhanyo, Z. Kalmar, F. Shi, A. Narayanan und G. Karsai. „Reusable Idioms and Patterns in Graph Transformation Languages“. In: *Electronic Notes in Theoretical Computer Science* 127.1 (2005). Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004), S. 181–192. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2004.12.035 (siehe S. 116).

[AK03] C. Atkinson und T. Kühne. „Model-driven development: a metamodeling foundation“. In: *Software, IEEE* 20.5 (2003), S. 36–41. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231149 (siehe S. 16, 20, 21).

[ALB08] M. F. van Amstel, C. F. J. Lange und M. G. J. van den Brand. „Metrics for analyzing the quality of model transformations“. In: *Proceedings 12th ECOOP Workshop on Quantitative Approaches on Object Oriented Software Engineering (QAOOSE08, Paphos, Cyprus, July 8, 2008 (co-located with ECOOP 2008))*. Hrsg. von G. Falcone, Y. G. Guéhéneuc, C. F. J. Lange, Z. Porkoláb und H. A. Sahraoui. Juli 2008, S. 41–51 (siehe S. 114, 115).

[ALB09] M. F. van Amstel, C. F. J. Lange und M. G. J. van den Brand. „Using Metrics for Assessing the Quality of ASF+SDF Model Transformations“. In: *Theory and Practice of Model Transformations*. Hrsg. von R. F. Paige. Bd. 5563. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 239–248. ISBN: 978-3-642-02407-8. DOI: 10.1007/978-3-642-02408-5_17 (siehe S. 115).

[Ali+11] S. Ali, M. Iqbal, A. Arcuri und L. Briand. „A Search-Based OCL Constraint Solver for Model-Based Test Data Generation“. In: *Quality Software (QSIC), 2011 11th International Conference on*. 2011, S. 41–50. DOI: 10.1109/QSIC.2011.17 (siehe S. 106).

[All70] F. E. Allen. „Control Flow Analysis“. In: *Proceedings of a Symposium on Compiler Optimization*. Urbana-Champaign, Illinois, USA: ACM, 1970, S. 1–19. DOI: 10.1145/800028.808479 (siehe S. 60, 207).

[AM71] E. A. Ashcroft und Z. Manna. *The Translation of "Go to" Programs to "While" Programs*. Techn. Ber. Stanford, Kalifornien, USA, 1971 (siehe S. 60).

[Ame+06] C. Amelunxen, A. Königs, T. Rötschke und A. Schürr. „MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations“. In: *Model Driven Architecture – Foundations and Applications*. Hrsg. von A. Rensink und J. Warmer. Bd. 4066. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 361–375. ISBN: 978-3-540-35909-8. DOI: 10.1007/11787044_27 (siehe S. 22, 36, 53).

[Amr+12] M. Amrani, L. Lucio, G. Selim, B. Combemale, J. Dingel, H. Vangheluwe, Y. Le-Traon und J. R. Cordy. „A Tridimensional Approach for Studying the Formal Verification of Model Transformations“. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. Apr. 2012, S. 921–928. DOI: 10.1109/ICST.2012.197 (siehe S. 4, 117).

[Ams10] M. F. van Amstel. „The Right Tool for the Right Job: Assessing Model Transformation Quality“. In: *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*. Juli 2010, S. 69–74. DOI: 10.1109/COMPSACW.2010.22 (siehe S. 114).

[Ana+07] K. Anastasakis, B. Bordbar, G. Georg und I. Ray. „UML2Alloy: A Challenging Model Transformation“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von G. Engels, B. Opdyke, D. Schmidt und F. Weil. Bd. 4735. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 436–450. ISBN: 978-3-540-75208-0. DOI: 10.1007/978-3-540-75209-7_30 (siehe S. 112, 118, 286).

[And+03] A. Andrews, R. France, S. Ghosh und G. Craig. „Test adequacy criteria for UML design models“. In: *Software Testing, Verification and Reliability* 13.2 (2003), S. 95–127. ISSN: 1099-1689. DOI: 10.1002/stvr.270 (siehe S. 108, 109, 125).

[And+06] J. H. Andrews, L. C. Briand, Y. Labiche und A. S. Namin. „Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria“. In: *Software Engineering, IEEE Transactions on* 32.8 (Aug. 2006), S. 608–624. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.83 (siehe S. 9, 87, 197, 199).

[And+99] M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Plump, A. Schürr und G. Taentzer. „Graph transformation for specification and programming“. In: *Science of Computer Programming* 34.1 (1999), S. 1–54. ISSN: 0167-6423. DOI: 10.1016/S0167-6423(98)00023-9 (siehe S. 38, 42, 44, 48).

[Anj+11] A. Anjorin, M. Lauder, S. Patzina und A. Schürr. „eMoflon: Leveraging EMF and Professional CASE Tools“. In: *3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011)*. Bonn: Gesellschaft für Informatik, 2011 (siehe S. 53).

[AO00] A. Abdurazik und J. Offutt. „Using UML Collaboration Diagrams for Static Checking and Test Generation“. In: *«UML»2000 – The Unified Modeling Language*. Hrsg. von A. Evans, S. Kent und B. Selic. Bd. 1939. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, S. 383–395. ISBN: 978-3-540-41133-8. DOI: 10.1007/3-540-40011-7_28 (siehe S. 202).

[AO08] P. Ammann und J. Offutt. *Introduction to Software Testing*. 1. Aufl. New York, NY, USA: Cambridge University Press, 2008. ISBN: 978-0-521-88038-1 (siehe S. 9, 80, 84–87, 89, 91–101, 109, 111, 129, 186, 198, 228, 232, 240, 284, 285).

[AP03] M. Alanen und I. Porres. „Difference and Union of Models“. In: *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*. Hrsg. von P. Stevens, J. Whittle und G. Booch. Bd. 2863. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, S. 2–17. ISBN: 978-3-540-20243-1. DOI: 10.1007/978-3-540-45221-8_2 (siehe S. 132).

[Ara+10] V. Aranega, J.-M. Mottu, A. Etien und J. Dekeyser. „Using Traceability to Enhance Mutation Analysis Dedicated to Model Transformation“. In: *Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010 Workshop on*. 2010, S. 1–6. DOI: 10.1109/MoDeVVa.2010.15 (siehe S. 129).

[Are+14] T. Arendt, A. Habel, H. Radke und G. Taentzer. „From Core OCL Invariants to Nested Graph Constraints“. In: *Graph Transformation*. Hrsg. von H. Giese und B. König. Bd. 8571. Lecture Notes in Computer Science. Springer International Publishing, 2014, S. 97–112. ISBN: 978-3-319-09107-5. DOI: 10.1007/978-3-319-09108-2_7 (siehe S. 113).

[AS05] B. K. Aichernig und P. A. P. Salas. „Test case generation by OCL mutation and constraint solving“. In: *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*. Sep. 2005, S. 64–71. DOI: 10.1109/QSIC.2005.63 (siehe S. 110).

[AS08] C. Amelunxen und A. Schürr. „Formalising model transformation rules for UML/MOF 2“. In: *Software, IET* 2.3 (Juni 2008), S. 204–222. ISSN: 1751-8806. DOI: 10.1049/iet-sen:20070076 (siehe S. 25).

[AVS12] A. Anjorin, G. Varró und A. Schürr. „Complex Attribute Manipulation in TGGs with Constraint-Based Programming Techniques“. In: *ECEASST* 49 (2012). Hrsg. von F. Hermann und J. Voigtländer. http://journal.ub.tu-berlin.de/eceasst/article/view/707. European Assoc. of Software Science and Technology (siehe S. 68).

[AW13] L. Ab Rahim und J. Whittle. „A survey of approaches for verifying model transformations“. In: *Software & Systems Modeling* (2013), S. 1–26. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0358-0 (siehe S. 115, 117, 120, 124, 131).

[BA82] T. A. Budd und D. Angluin. „Two notions of correctness and their relation to testing“. In: *Acta Informatica* 18.1 (1982), S. 31–45. ISSN: 0001-5903. DOI: 10.1007/BF00625279 (siehe S. 99).

[Bal+06] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits und S. Neema. „Developing applications using model-driven design environments“. In: *Computer* 39.2 (2006), S. 33–40. ISSN: 0018-9162. DOI: 10.1109/MC.2006.54 (siehe S. 15).

[Bal98] H. Balzert. *Lehrbuch der Software-Technik : Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Bd. 2. Spektrum, Akademischer Verlag, 1998. ISBN: 3-8274-0065-1 (siehe S. 4, 6, 80, 82).

[Bar+03] A. Baresel, M. Conrad, S. Sadeghipour und J. Wegener. „The interplay between model coverage and code coverage“. In: *Proc. EuroCAST*. 2003 (siehe S. 136, 190).

[Bau+06] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey und Y. Le Traon. „Model Transformation Testing Challenges“. In: *ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing*. Bilbao, Spanien, 2006 (siehe S. 5, 124, 126, 127, 131, 133, 137).

[Bau+10] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon und J.-M. Mottu. „Barriers to Systematic Model Transformation Testing“. In: *Commun. ACM* 53.6 (2010), S. 139–143. ISSN: 0001-0782. DOI: 10.1145/1743546.1743583 (siehe S. 5, 124, 137).

[BBL76] B. W. Boehm, J. R. Brown und M. Lipow. „Quantitative Evaluation of Software Quality“. In: *Proceedings of the 2nd International Conference on Software Engineering*. ICSE '76. San Francisco, Kalifornien, USA: IEEE Computer Society Press, 1976, S. 592–605 (siehe S. 114).

[BCG05] D. Berardi, D. Calvanese und G. D. Giacomo. „Reasoning on UML class diagrams“. In: *Artificial Intelligence* 168.1–2 (2005), S. 70–118. ISSN: 0004-3702. DOI: 10.1016/j.artint.2005.05.003 (siehe S. 112).

[BCG12] M. Broy, M. V. Cengarle und E. Geisberger. „Cyber-Physical Systems: Imminent Challenges“. In: *Large-Scale Complex IT Systems. Development, Operation and Management*. Hrsg. von R. Calinescu und D. Garlan. Bd. 7539. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 1–28. ISBN: 978-3-642-34058-1. DOI: 10.1007/978-3-642-34059-8_1 (siehe S. 2).

[BCK01] P. Baldan, A. Corradini und B. König. „A Static Analysis Technique for Graph Transformation Systems“. In: *CONCUR 2001 – Concurrency Theory*. Hrsg. von K. G. Larsen und M. Nielsen. Bd. 2154. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, S. 381–395. ISBN: 978-3-540-42497-0. DOI: 10.1007/3-540-44685-0_26 (siehe S. 121).

[BCL03] L. C. Briand, J. Cui und Y. Labiche. „Towards Automated Support for Deriving Test Data from UML Statecharts“. In: *«UML» 2003 - The Unified Modeling Language. Modeling Languages and Applications*. Hrsg. von P. Stevens, J. Whittle und G. Booch. Bd. 2863. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, S. 249–264. ISBN: 978-3-540-20243-1. DOI: 10.1007/978-3-540-45221-8_22 (siehe S. 107).

[BCW12] M. Brambilla, J. Cabot und M. Wimmer. *Model-Driven Software Engineering in Practice*. Hrsg. von D. Cerra. Synthesis Lectures on Software Engineering. Morgan und Claypool, Sep. 2012, S. 182. DOI: 10.2200/S00441ED1V01Y201208SWE001 (siehe S. 15).

[BEC12] F. Büttner, M. Egea und J. Cabot. „On Verifying ATL Transformations Using ‘off-the-shelf’ SMT Solvers“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von R. B. France, J. Kazmeier, R. Breu und C. Atkinson. Bd. 7590. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 432–448. ISBN: 978-3-642-33665-2. DOI: 10.1007/978-3-642-33666-9_28 (siehe S. 118).

[BEH07] L. Baresi, K. Ehrig und R. Heckel. „Verification of Model Transformations: A Case Study with BPEL“. In: *Trustworthy Global Computing*. Hrsg. von U. Montanari, D. Sannella und R. Bruni. Bd. 4661. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 183–199. ISBN: 978-3-540-75333-9. DOI: 10.1007/978-3-540-75336-0_12 (siehe S. 120).

[Bei90] B. Beizer. *Software testing techniques (2. ed.)* Van Nostrand Reinhold, 1990, S. I–XXVI, 1–550. ISBN: 978-0-442-20672-7 (siehe S. 80, 82, 91, 145).

[BET08] E. Biermann, C. Ermel und G. Taentzer. „Precise Semantics of EMF Model Transformations by Graph Transformation“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl und M. Völter. Bd. 5301. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 53–67. ISBN: 978-3-540-87874-2. DOI: 10.1007/978-3-540-87875-9_4 (siehe S. 24, 25, 48, 165).

[BET12] E. Biermann, C. Ermel und G. Taentzer. „Formal foundation of consistent EMF model transformations by algebraic graph transformation“. In: *Software & Systems Modeling* 11.2 (2012), S. 227–250. ISSN: 1619-1366. DOI: 10.1007/s10270-011-0199-7 (siehe S. 48).

[Béz+06a] J. Bézivin, F. Büttner, M. Gogolla, F. Jouault, I. Kurtev und A. Lindow. „Model Transformations? Transformation Models!“ In: *Model Driven Engineering Languages and Systems*. Hrsg. von O. Nierstrasz, J. Whittle, D. Harel und G. Reggio. Bd. 4199. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 440–453. ISBN: 978-3-540-45772-5. DOI: 10.1007/11880240_31 (siehe S. 38).

[Béz+06b] J. Bézivin, B. Rumpe, A. Schürr und L. Tratt. „Model Transformations in Practice Workshop“. In: *Satellite Events at the MoDELS 2005 Conference*. Hrsg. von J.-M. Bruel. Bd. 3844. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 120–127. ISBN: 978-3-540-31780-7. DOI: 10.1007/11663430_13 (siehe S. 34).

[Béz04] J. Bézivin. „In Search of a Basic Principle for Model Driven Engineering“. In: *CEPIS UPGRADE The European Journal for the Informatics Professional* V.2 (Apr. 2004), S. 21–24. ISSN: 1684-5285 (siehe S. 19–21, 23).

[Béz05] J. Bézivin. „On the unification power of models“. In: *Software & Systems Modeling* 4.2 (2005), S. 171–188. ISSN: 1619-1366. DOI: 10.1007/s10270-005-0079-0 (siehe S. 20, 21).

[BG01] J. Bézivin und O. Gerbe. „Towards a precise definition of the OMG/MDA framework“. In: *Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on*. Nov. 2001, S. 273–280. DOI: 10.1109/ASE.2001.989813 (siehe S. 21).

[BG06] F. Büttner und M. Gogolla. „Realizing Graph Transformations by Pre- and Postconditions and Command Sequences“. In: *Graph Transformations*. Hrsg. von A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro und G. Rozenberg. Bd. 4178. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 398–413. ISBN: 978-3-540-38870-8. DOI: 10.1007/11841883_28 (siehe S. 122).

[BGM91] G. Bernot, M.-C. Gaudel und B. Marre. „Software testing based on formal specifications: a theory and a tool“. In: *Software Engineering Journal* 6.6 (Nov. 1991), S. 387–405. ISSN: 0268-6961 (siehe S. 106).

[BH02] L. Baresi und R. Heckel. „Tutorial Introduction to Graph Transformation: A Software Engineering Perspective“. In: *Graph Transformation*. Hrsg. von A. Corradini, H. Ehrig, H.-J. Kreowski und G. Rozenberg. Bd. 2505. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, S. 402–429. ISBN: 978-3-540-44310-0. DOI: 10.1007/3-540-45832-8_30 (siehe S. 42, 44).

[BHM09] A. Boronat, R. Heckel und J. Meseguer. „Rewriting Logic Semantics and Verification of Model Transformations“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von M. Chechik und M. Wirsing. Bd. 5503. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 18–33. ISBN: 978-3-642-00592-3. DOI: 10.1007/978-3-642-00593-0_2 (siehe S. 119).

[Bic03] L. Bichler. „A flexible code generator for MOF-based modeling languages“. In: *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. 2003*. 2003 (siehe S. 36).

[Bie+10a] E. Biermann, H. Ehrig, C. Ermel, U. Golas und G. Taentzer. „Parallel Independence of Amalgamated Graph Transformations Applied to Model Transformation“. In: *Graph Transformations and Model-Driven Engineering*. Hrsg. von G. Engels, C. Lewerentz, W. Schäfer, A. Schürr und B. Westfechtel. Bd. 5765. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 121–140. ISBN: 978-3-642-17321-9. DOI: 10.1007/978-3-642-17322-6_7 (siehe S. 54).

[Bie+10b] E. Biermann, C. Ermel, L. Lambers, U. Prange, O. Runge und G. Taentzer. „Introduction to AGG and EMF Tiger by modeling a Conference Scheduling System“. In: *International Journal on Software Tools for Technology Transfer* 12.3-4 (2010), S. 245–261. ISSN: 1433-2779. DOI: 10.1007/s10009-010-0154-x (siehe S. 46).

[Bin99] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-80938-9 (siehe S. 88, 106, 201, 202).

[BJ66] C. Böhm und G. Jacopini. „Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules“. In: *Commun. ACM* 9.5 (Mai 1966), S. 366–371. ISSN: 0001-0782. DOI: 10.1145/355592.365646 (siehe S. 60, 207).

[BJP05] J. Bézivin, F. Jouault und J. Paliès. „Towards model transformation design patterns“. In: *Proceedings of the First European Workshop on Model Transformations (EWMT 2005)*. 2005 (siehe S. 116).

[BKE11] E. Bauer, J. M. Küster und G. Engels. „Test Suite Quality for Model Transformation Chains“. In: *Objects, Models, Components, Patterns*. Hrsg. von J. Bishop und A. Vallecillo. Bd. 6705. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 3–19. ISBN: 978-3-642-21951-1. DOI: 10.1007/978-3-642-21952-8_3 (siehe S. 127).

[BKM02] C. Boyapati, S. Khurshid und D. Marinov. „Korat: Automated Testing Based on Java Predicates“. In: *SIGSOFT Softw. Eng. Notes* 27.4 (Juli 2002), S. 123–133. ISSN: 0163-5948. DOI: 10.1145/566171.566191 (siehe S. 125).

[BKS02] B. Beckert, U. Keller und P. H. Schmitt. „Translating the Object Constraint Language into First-order Predicate Logic“. In: *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Kopenhagen, Dänemark*. 2002 (siehe S. 112).

[BKS04] P. Baldan, B. König und I. Stürmer. „Generating Test Cases for Code Generators by Unfolding Graph Transformation Systems“. In: *Graph Transformations*. Hrsg. von H. Ehrig, G. Engels, F. Parisi-Presicce und G. Rozenberg. Bd. 3256. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, S. 194–209. ISBN: 978-3-540-23207-0. DOI: 10.1007/978-3-540-30203-2_15 (siehe S. 6, 125, 134, 136, 138, 196).

[BLL04] X. Bai, C. P. Lam und H. Li. „An Approach to Generate the Thin-threads from the UML Diagrams“. In: *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. Sep. 2004, 546–552 vol.1. DOI: 10.1109/CMPSAC.2004.1342893 (siehe S. 110, 111).

[BMT11] M. Broy, K. H. Mühleck und D. Taubner. „Informatik in der Automobilindustrie“. In: *Informatik-Spektrum* 34.1 (2011), S. 1–5. ISSN: 0170-6012. DOI: 10.1007/s00287-010-0508-5 (siehe S. 1).

[Boe79] B. W. Boehm. „Guidelines for Verifying and Validating Software Requirements and Design Specifications“. In: *Proceedings of Euro IFIP 79*. Hrsg. von P. A. Samet. North-Holland Publishing Company, 1979 (siehe S. 2).

[Boe84] B. W. Boehm. „Verifying and Validating Software Requirements and Design Specifications“. In: *Software, IEEE* 1.1 (Jan. 1984), S. 75–88. ISSN: 0740-7459. DOI: 10.1109/MS.1984.233702 (siehe S. 4).

[Bot+05] P. Bottoni, M. Koch, F. Parisi-Presicce und G. Taentzer. „Termination of High-Level Replacement Units with Application to Model Transformation“. In: *Electronic Notes in Theoretical Computer Science* 127.4 (2005). Proceedings of the Workshop on Visual Languages and Formal Methods (VLFM 2004), S. 71–86. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2004.08.048 (siehe S. 121).

[BP08] C. Brun und A. Pierantonio. „Model Differences in the Eclipse Modelling Framework“. In: *CEPIS UPGRADE The European Journal for the Informatics Professional* IX (Apr. 2008), S. 29–34. ISSN: 1684-5285 (siehe S. 132).

[BR70] J. N. Buxton und B. Randell, Hrsg. *Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*. Brüssel: NATO Scientific Affairs Div., Apr. 1970 (siehe S. 79).

[Bra+11] C. Braga, R. Menezes, T. Comicio, C. Santos und E. Landim. „On the Specification, Verification and Implementation of Model Transformations with Transformation Contracts“. In: *Formal Methods, Foundations and Applications*. Hrsg. von A. Simao und C. Morgan. Bd. 7021. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 108–123. ISBN: 978-3-642-25031-6. DOI: 10.1007/978-3-642-25032-3_8 (siehe S. 119).

[Bro+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker und A. Pretschner, Hrsg. *Model-Based Testing of Reactive Systems*. Springer, 2005. ISBN: 978-3-540-26278-7 (siehe S. 102, 103).

[Bro+06] E. Brottier, F. Fleurey, J. Steel, B. Baudry und Y. Le Traon. „Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool“. In: *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*. 2006, S. 85–94. DOI: 10.1109/ISSRE.2006.27 (siehe S. 129).

[Bro+10] M. Broy, G. Schütte, H. Fischer, K. Beetz, W. Damm, R. Achatz, H. Daembkes, K. Grimm und P. Liggesmeyer. *Cyber-Physical Systems: Innovation durch softwareintensive eingebettete System*. Hrsg. von M. Broy. acatech – Deutsche Akademie der Technikwissenschaften, Springer-Verlag Berlin Heidelberg, 2010, S. 7–141. ISBN: 978-3-642-14498-1. DOI: 10.1007/978-3-642-14901-6 (siehe S. 2).

[Bru+11] A. D. Brucker, M. P. Krieger, D. Longuet und B. Wolff. „A Specification-Based Test Case Generation Method for UML/OCL“. In: *Models in Software Engineering*. Hrsg. von J. Dingel und A. Solberg. Bd. 6627. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 334–348. ISBN: 978-3-642-21209-3. DOI: 10.1007/978-3-642-21210-9_33 (siehe S. 113).

[BS03] E. Börger und R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Berlin Heidelberg, 2003, S. I–X, 1–438. ISBN: 978-3-642-62116-1. DOI: 10.1007/978-3-642-18216-7 (siehe S. 333).

[BS06] L. Baresi und P. Spoletini. „On the Use of Alloy to Analyze Graph Transformation Systems“. In: *Graph Transformations*. Hrsg. von A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro und G. Rozenberg. Bd. 4178. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 306–320. ISBN: 978-3-540-38870-8. DOI: 10.1007/11841883_22 (siehe S. 122).

[BSR10] D. Benavides, S. Segura und A. Ruiz-Cortés. „Automated Analysis of Feature Models 20 Years Later: A Literature Review“. In: *Inf. Syst.* 35.6 (Sep. 2010), S. 615–636. ISSN: 0306-4379. DOI: 10.1016/j.is.2010.01.001 (siehe S. 32, 33).

[Bun] Bundesministerium für Bildung und Forschung, Referat IT-Systeme. *Zukunftsbild „Industrie 4.0“ – Hightech-Strategie*. http://www.bmbf.de/pubRD/Zukunftsbild_Industrie_40.pdf (zuletzt abgerufen am 16.9.2014) (siehe S. 2).

[Bun79] H. Bunke. „Programmed graph grammars“. In: *Graph-Grammars and Their Application to Computer Science and Biology*. Hrsg. von V. Claus, H. Ehrig und G. Rozenberg. Bd. 73. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1979, S. 155–166. ISBN: 978-3-540-09525-5. DOI: 10.1007/BFb0025718 (siehe S. 52).

[Bur+05] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino und E. Poll. „An overview of JML tools and applications“. In: *International Journal on Software Tools for Technology Transfer* 7.3 (2005), S. 212–232. ISSN: 1433-2779. DOI: 10.1007/s10009-004-0167-4 (siehe S. 106).

[Büt+12a] F. Büttner, M. Egea, J. Cabot und M. Gogolla. „Verification of ATL Transformations Using Transformation Models and Model Finders“. In: *Formal Methods and Software Engineering*. Hrsg. von T. Aoki und K. Taguchi. Bd. 7635. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 198–213. ISBN: 978-3-642-34280-6. DOI: 10.1007/978-3-642-34281-3_16 (siehe S. 118).

[Büt+12b] F. Büttner, M. Egea, J. Cabot und M. Gogolla. „Verification of ATL Transformations Using Transformation Models and Model Finders“. In: *Formal Methods and Software Engineering*. Hrsg. von T. Aoki und K. Taguchi. Bd. 7635. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 198–213. ISBN: 978-3-642-34280-6. DOI: 10.1007/978-3-642-34281-3_16 (siehe S. 133).

[BW08] A. D. Brucker und B. Wolff. „HOL-OCL: A Formal Proof Environment for UML/OCL“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von J. L. Fiadeiro und P. Inverardi. Bd. 4961. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 97–100. ISBN: 978-3-540-78742-6. DOI: 10.1007/978-3-540-78743-3_8 (siehe S. 113).

[BW09] A. D. Brucker und B. Wolff. „HOL-TestGen“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von M. Chechik und M. Wirsing. Bd. 5503. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 417–420. ISBN: 978-3-642-00592-3. DOI: 10.1007/978-3-642-00593-0_28 (siehe S. 113).

[BW13] A. D. Brucker und B. Wolff. „On theorem prover-based testing“. In: *Formal Aspects of Computing* 25.5 (2013), S. 683–721. ISSN: 0934-5043. DOI: 10.1007/s00165-012-0222-y (siehe S. 106).

[BWW12] T. Buchmann, B. Westfechtel und S. Winetzhammer. „The Added Value of Programmed Graph Transformations – A Case Study from Software Configuration Management“. In: *Applications of Graph Transformations with Industrial Relevance*. Hrsg. von A. Schürr, D. Varró und G. Varró. Bd. 7233. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 198–209. ISBN: 978-3-642-34175-5. DOI: 10.1007/978-3-642-34176-2_17 (siehe S. 247, 248).

[BY01] L. Baresi und M. Young. *Test Oracles*. Technical Report CIS-TR-01-02. http://www.cs.uoregon.edu/~michal/pubs/oracles.html. Eugene, Oregon, USA: University of Oregon, Dept. of Computer und Information Science, Aug. 2001 (siehe S. 90).

[Cab+08] J. Cabot, R. Clarisó, E. Guerra und J. de Lara. „Analysing Graph Transformation Rules through OCL“. In: *Theory and Practice of Model Transformations*. Hrsg. von A. Vallecillo, J. Gray und A. Pierantonio. Bd. 5063. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 229–244. ISBN: 978-3-540-69926-2. DOI: 10.1007/978-3-540-69927-9_16 (siehe S. 121).

[Cab+10a] J. Cabot, R. Clarisó, E. Guerra und J. de Lara. „A UML/OCL framework for the analysis of graph transformation rules“. In: *Software & Systems Modeling* 9.3 (2010), S. 335–357. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0129-0 (siehe S. 121, 130).

[Cab+10b] J. Cabot, R. Clarisó, E. Guerra und J. de Lara. „Synthesis of OCL Preconditions for Graph Transformation Rules“. In: *Theory and Practice of Model Transformations*. Hrsg. von L. Tratt und M. Gogolla. Bd. 6142. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 45–60. ISBN: 978-3-642-13687-0. DOI: 10.1007/978-3-642-13688-7_4 (siehe S. 123).

[Cab+10c] J. Cabot, R. Clarisó, E. Guerra und J. de Lara. „Verification and Validation of Declarative Model-to-model Transformations Through Invariants“. In: *Journal of Systems and Software* 83.2 (Feb. 2010), S. 283–302. ISSN: 0164-1212. DOI: 10.1016/j.jss.2009.08.012 (siehe S. 118, 133).

[Car+04a] E. Cariou, R. Marvie, L. Seinturier und L. Duchien. *OCL for the Specification of Model Transformation Contracts*. http://www.cs.kent.ac.uk/projects/ocl/oclmdewsuml04/. Workshop OCL and Model Driven Engineering of the Seventh International Conference on UML Modeling Languages and Applications (UML 2004). 2004 (siehe S. 125–127).

[Car+04b] E. Cariou, R. Marvie, L. Seinturier und L. Duchien. *Model Transformation Contracts and their Definition in UML and OCL*. Techn. Ber. 2004-08. Laboratoire d'Informatique Fondamentale de Lille (LIFL), 2004 (siehe S. 125–127).

[Car+09] E. Cariou, N. Belloir, F. Barbier und N. Djemam. „OCL Contracts for the Verification of Model Transformations". In: *ECEASST* 24 (2009). Hrsg. von J. Cabot, J. Chimiak-Opoka, F. Jouault, M. Gogolla und A. Knapp. `http://journal.ub.tu-berlin.de/eceasst/article/view/326`. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 125, 133).

[CCR07] J. Cabot, R. Clarisó und D. Riera. „UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming". In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. ASE '07. Atlanta, Georgia, USA: ACM, 2007, S. 547–548. ISBN: 978-1-59593-882-4. DOI: `10.1145/1321631.1321737` (siehe S. 112, 129).

[CCR08] J. Cabot, R. Clarisó und D. Riera. „Verification of UML/OCL Class Diagrams using Constraint Programming". In: *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*. Apr. 2008, S. 73–80. DOI: `10.1109/ICSTW.2008.54` (siehe S. 113).

[CFM10] A. Ciancone, A. Filieri und R. Mirandola. „MANTra: Towards Model Transformation Testing". In: *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. 2010, S. 97–105. DOI: `10.1109/QUATIC.2010.15` (siehe S. 128).

[CFM13] A. Ciancone, A. Filieri und R. Mirandola. „Testing operational transformations in model-driven engineering". In: *Innovations in Systems and Software Engineering* Online First Article (2013), S. 1–14. ISSN: 1614-5046. DOI: `10.1007/s11334-013-0208-9` (siehe S. 128).

[CH03] K. Czarnecki und S. Helsen. „Classification of Model Transformation Approaches". In: *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*. Anaheim, Kalifornien, USA, 2003 (siehe S. 31, 32).

[CH06] K. Czarnecki und S. Helsen. „Feature-based survey of model transformation approaches". In: *IBM Systems Journal* 45.3 (2006), S. 621–645. ISSN: 0018-8670. DOI: `10.1147/sj.453.0621` (siehe S. 30–32).

[Che+05a] K. Chen, J. Sztipanovits, S. Abdelwalhed und E. Jackson. „Semantic Anchoring with Model Transformations". In: *Model Driven Architecture – Foundations and Applications*. Hrsg. von A. Hartman und D. Kreische. Bd. 3748. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 115–129. ISBN: 978-3-540-30026-7. DOI: `10.1007/11581741_10` (siehe S. 4).

[Che+05b] T. Y. Chen, P.-L. Poon, S.-F. Tang und T. H. Tse. „Identification of Categories and Choices in Activity Diagrams". In: *Quality Software, 2005. (QSIC 2005). Fifth International Conference on*. Sep. 2005, S. 55–63. DOI: `10.1109/QSIC.2005.36` (siehe S. 111).

[Che+09] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao und X. Li. „UML Activity Diagram-Based Automatic Test Case Generation For Java Programs“. In: *The Computer Journal* 52.5 (2009), S. 545–556. DOI: 10.1093/comjnl/bxm057 (siehe S. 110, 111).

[Che01] P. Chevalley. „Applying mutation analysis for object-oriented programs using a reflective approach“. In: *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*. Dez. 2001, S. 267–270. DOI: 10.1109/APSEC.2001.991487 (siehe S. 201).

[Che76] P. P.-S. Chen. „The Entity-Relationship Model – Toward a Unified View of Data“. In: *ACM Transactions on Database Systems* 1.1 (März 1976), S. 9–36. ISSN: 0362-5915. DOI: 10.1145/320434.320440 (siehe S. 17).

[Cho78] T. S. Chow. „Testing Software Design Modeled by Finite-State Machines“. In: *Software Engineering, IEEE Transactions on* SE-4.3 (1978), S. 178–187. ISSN: 0098-5589. DOI: 10.1109/TSE.1978.231496 (siehe S. 93, 102, 105, 111).

[CL05] I. Ciupa und A. Leitner. „Automatic Testing Based on Design by Contract“. In: *Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World)*. Sep. 2005, S. 545–557 (siehe S. 125).

[Cla+02] D. Clarke, T. Jéron, V. Rusu und E. Zinovieva. „STG: A Symbolic Test Generation Tool“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von J.-P. Katoen und P. Stevens. Bd. 2280. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, S. 470–475. ISBN: 978-3-540-43419-1. DOI: 10.1007/3-540-46002-0_34 (siehe S. 106).

[Cla76] L. A. Clarke. „A System to Generate Test Data and Symbolically Execute Programs“. In: *Software Engineering, IEEE Transactions on* SE-2.3 (Sep. 1976), S. 215–222. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233817 (siehe S. 106, 129).

[CM09] M. Cristiá und P. Monetti. „Implementing and Applying the Stocks-Carrington Framework for Model-Based Testing“. In: *Formal Methods and Software Engineering*. Hrsg. von K. Breitman und A. Cavalcanti. Bd. 5885. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 167–185. ISBN: 978-3-642-10372-8. DOI: 10.1007/978-3-642-10373-5_9 (siehe S. 106).

[CM94] J. J. Chilenski und S. P. Miller. „Applicability of modified condition/decision coverage to software testing“. In: *Software Engineering Journal* 9.5 (Sep. 1994), S. 193–200. ISSN: 0268-6961 (siehe S. 334).

[CMK08] M. Chen, P. Mishra und D. Kalita. „Coverage-driven Automatic Test Generation for UML Activity Diagrams“. In: *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*. GLSVLSI '08. Orlando, Florida, USA: ACM, 2008, S. 139–142. ISBN: 978-1-59593-999-9. DOI: 10.1145/1366110.1366145 (siehe S. 110, 111).

[CMK10] M. Chen, P. Mishra und D. Kalita. „Efficient test case generation for validation of UML activity diagrams“. In: *Design Automation for Embedded Systems* 14.2 (2010), S. 105–130. ISSN: 0929-5585. DOI: 10.1007/s10617-010-9052-4 (siehe S. 110, 111).

[Cor+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel und M. Löwe. „Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach“. In: *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations*. World Scientific Publishing Co., Inc., 1997, S. 163–245 (siehe S. 46).

[CQL06] M. Chen, X. Qiu und X. Li. „Automatic Test Case Generation for UML Activity Diagrams“. In: *Proceedings of the 2006 International Workshop on Automation of Software Test*. AST '06. Shanghai, China: ACM, 2006, S. 2–8. ISBN: 1-59593-408-1. DOI: 10.1145/1138929.1138931 (siehe S. 111).

[CR09] S. A. da Costa und L. Ribeiro. „Formal Verification of Graph Grammars using Mathematical Induction“. In: *Electronic Notes in Theoretical Computer Science* 240 (2009). Proceedings of the Eleventh Brazilian Symposium on Formal Methods (SBMF 2008), S. 43–60. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2009.05.044 (siehe S. 123).

[CR12] S. A. da Costa und L. Ribeiro. „Verification of graph grammars using a logical approach“. In: *Science of Computer Programming* 77.4 (2012). Brazilian Symposium on Formal Methods (SBMF 2008), S. 480–504. ISSN: 0167-6423. DOI: 10.1016/j.scico.2010.02.006 (siehe S. 123).

[CS12] D. Calegari und N. Szasz. *Verification of Model Transformations: A Survey of the State-of-the-Art (Extended Version)*. Techn. Ber. RT 12-05. http://www.fing.edu.uy/inco/pedeciba/bibpm/field.php/Main/ReportesT%e9cnicos (zuletzt abgerufen am 29.9.2014). Instituto de Computacion – Facultad de Ingenieria, Universidad de la Republica Montevideo, Uruguay, Juni 2012 (siehe S. 117).

[CS13] D. Calegari und N. Szasz. „Verification of Model Transformations: A Survey of the State-of-the-Art“. In: *Electronic Notes in Theoretical Computer Science* 292 (2013). Proceedings of the XXXVIII Latin American Conference in Informatics (CLEI), S. 5–25. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2013.02.002 (siehe S. 117).

[Cse+02] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza und D. Varró. „VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models“. In: *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*. 2002, S. 267–270. DOI: 10.1109/ASE.2002.1115027 (siehe S. 51).

[Cua+09] J. S. Cuadrado, F. Jouault, J. García Molina und J. Bézivin. „Optimization Patterns for OCL-Based Model Transformations“. In: *Models in Software Engineering*. Hrsg. von M. R. V. Chaudron. Bd. 5421. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 273–284. ISBN: 978-3-642-01647-9. DOI: 10.1007/978-3-642-01648-6_29 (siehe S. 116).

[Dal+99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton und B. M. Horowitz. „Model-Based Testing in Practice“. In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE '99. Los Angeles, Kalifornien, USA: ACM, 1999, S. 285–294. ISBN: 1-58113-074-0. DOI: 10.1145/302405.302640 (siehe S. 102).

[Dan02] J. Daniels. „Modeling with a sense of purpose“. In: *Software, IEEE* 19.1 (Jan. 2002), S. 8–10. ISSN: 0740-7459. DOI: 10.1109/52.976934 (siehe S. 19).

[Dar07] A. Darabos. „Testing the Implementation of Graph Transformations“. Masterarbeit. Budapest University of Technology, Economics - Department of Measurement und Information Systems, Mai 2007 (siehe S. 134, 139).

[DeM+88] R. A. DeMillo, D. S. Guindi, W. M. McCracken, A. J. Offutt und K. N. King. „An extended overview of the Mothra software testing environment“. In: *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*. Juli 1988, S. 142–151. DOI: 10.1109/WST.1988.5369 (siehe S. 98, 198).

[DGF06] T. T. Dinh-Trong, S. Ghosh und R. B. France. „A Systematic Approach to Generate Inputs to Test UML Design Models“. In: *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*. Nov. 2006, S. 95–104. DOI: 10.1109/ISSRE.2006.10 (siehe S. 109).

[DH01] W. Damm und D. Harel. „LSCs: Breathing Life into Message Sequence Charts“. In: *Formal Methods in System Design* 19.1 (2001), S. 45–80. ISSN: 0925-9856. DOI: 10.1023/A:1011227529550 (siehe S. 247).

[DH81] A. G. Duncan und J. S. Hutchison. „Using Attributed Grammars to Test Designs and Implementations“. In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, Kalifornien, USA: IEEE Press, 1981, S. 170–178. ISBN: 0-89791-146-6 (siehe S. 97).

[Dia+07] A. C. Dias Neto, R. Subramanyan, M. Vieira und G. H. Travassos. „A Survey on Model-based Testing Approaches: A Systematic Review“. In: *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies: Held in Conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. WEASELTech '07. Atlanta, Georgia, USA: ACM, 2007, S. 31–36. ISBN: 978-1-59593-880-0. DOI: 10.1145/1353673.1353681 (siehe S. 6, 102, 106).

[Dij68] E. W. Dijkstra. „Letters to the Editor: Go To Statement Considered Harmful“. In: *Commun. ACM* 11.3 (März 1968), S. 147–148. ISSN: 0001-0782. DOI: 10.1145/362929.362947 (siehe S. 60, 206).

[Din+05] T. Dinh-Trong, S. Ghosh, R. France, B. Baudry und F. Fleurey. „A Taxonomy of Faults for UML Designs“. In: *Proceedings of the MoDeVa Workshop at MODELS'05*. http://www.cs.colostate.edu/pubserv/pubs/Dinh-Trong-trungdt-publication-Dinh-FaultTaxonomy.pdf. Montego Bay, Jamaika, Oktober 2005 (siehe S. 108, 202).

[DL00] A. Dupuy und N. Leveson. „An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software“. In: *Digital Avionics Systems Conference, 2000. Proceedings. DASC. The 19th*. Bd. 1. 2000, 1B6/1–1B6/7 vol.1. DOI: 10.1109/DASC.2000.886883 (siehe S. 5).

[DLS78] R. A. DeMillo, R. J. Lipton und F. G. Sayward. „Hints on Test Data Selection: Help for the Practicing Programmer“. In: *Computer* 11.4 (Apr. 1978), S. 34–41. ISSN: 0018-9162. DOI: 10.1109/C-M.1978.218136 (siehe S. 8, 98, 101, 129, 151, 200).

[DM79] N. Dershowitz und Z. Manna. „Proving Termination with Multiset Orderings“. In: *Commun. ACM* 22.8 (Aug. 1979), S. 465–476. ISSN: 0001-0782. DOI: 10.1145/359138.359142 (siehe S. 121).

[DO91] R. A. DeMillo und A. J. Offutt. „Constraint-based automatic test data generation“. In: *Software Engineering, IEEE Transactions on* 17.9 (Sep. 1991), S. 900–910. ISSN: 0098-5589. DOI: 10.1109/32.92910 (siehe S. 265).

[DPV08] A. Darabos, A. Pataricza und D. Varró. „Towards Testing the Implementation of Graph Transformations“. In: *Electronic Notes in Theoretical Computer Science* 211 (2008). Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006), S. 75–85. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.031 (siehe S. 5, 6, 10, 124, 125, 129, 133, 134, 139, 146, 162, 205).

[Dre+06] F. Drewes, B. Hoffmann, D. Janssens, M. Minas und N. Van Eetvelde. „Adaptive Star Grammars“. In: *Graph Transformations*. Hrsg. von A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro und G. Rozenberg. Bd. 4178. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 77–91. ISBN: 978-3-540-38870-8. DOI: 10.1007/11841883_7 (siehe S. 130).

[DS07] A. Derezińska und A. Szustek. „CREAM - A System for Object-Oriented Mutation of C# Programs“. In: *Annals Gdańsk University of Technology Faculty of ETI (Information Technologies)*. 5. Danzig, 2007, S. 389–406 (siehe S. 198).

[DV14a] F. Deckwerth und G. Varró. „Attribute Handling for Generating Preconditions from Graph Constraints“. In: *Graph Transformation*. Hrsg. von H. Giese und B. König. Bd. 8571. Lecture Notes in Computer Science. Springer International Publishing, 2014, S. 81–96. ISBN: 978-3-319-09107-5. DOI: 10.1007/978-3-319-09108-2_6 (siehe S. 123).

[DV14b] F. Deckwerth und G. Varró. „Generating Preconditions from Graph Constraints by Higher Order Graph Transformation“. In: *ECEASST* 67 (Apr. 2014). Hrsg. von F. Hermann und S. Sauer. http://journal.ub.tu-berlin.de/eceasst/article/view/945. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 123).

[Eck03] C. Eckert. *IT-Sicherheit - Konzepte, Verfahren, Protokolle (2. Aufl.)* Oldenbourg, 2003. ISBN: 978-3-486-27205-5 (siehe S. 79).

[EE08] H. Ehrig und C. Ermel. „Semantical Correctness and Completeness of Model Transformations Using Graph and Rule Transformation“. In: *Graph Transformations*. Hrsg. von H. Ehrig, R. Heckel, G. Rozenberg und G. Taentzer. Bd. 5214. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 194–210. ISBN: 978-3-540-87404-1. DOI: 10.1007/978-3-540-87405-8_14 (siehe S. 120).

[EFM97] A. Engels, L. Feijs und S. Mauw. „Test generation for intelligent networks using model checking“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von E. Brinksma. Bd. 1217. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, S. 384–398. ISBN: 978-3-540-62790-6. DOI: 10.1007/BFb0035401 (siehe S. 106).

[Ehr+04] H. Ehrig, A. Habel, J. Padberg und U. Prange. „Adhesive High-Level Replacement Categories and Systems“. In: *Graph Transformations*. Hrsg. von H. Ehrig, G. Engels, F. Parisi-Presicce und G. Rozenberg. Bd. 3256. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, S. 144–160. ISBN: 978-3-540-23207-0. DOI: 10.1007/978-3-540-30203-2_12 (siehe S. 121).

[Ehr+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró und S. Varró-Gyapay. „Termination Criteria for Model Transformation“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von M. Cerioli. Bd. 3442. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 49–63. ISBN: 978-3-540-25420-1. DOI: 10.1007/978-3-540-31984-9_5 (siehe S. 121).

[Ehr+06] H. Ehrig, K. Ehrig, U. Prange und G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. Springer, 2006. ISBN: 978-3-540-31187-4. DOI: 10.1007/3-540-31188-2 (siehe S. 25, 39, 42, 44–48, 106).

[Ehr+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner und A. Corradini. „Algebraic Approaches to Graph Transformation, Part II: Single Pushout Aapproach and Comparison with Double Pushout Approach“. In: *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations*. World Scientific Publishing Co., Inc., 1997, S. 247–312 (siehe S. 46).

[Ehr+99a] H. Ehrig, G. Engels, H.-J. Kreowski und G. Rozenberg, Hrsg. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 2: Applications, Languages, and Tools*. River Edge, New Jersey, USA: World Scientific Publishing Co., Inc., 1999. ISBN: 981-02-4020-1 (siehe S. 42).

[Ehr+99b] H. Ehrig, H.-J. Kreowski, U. Montanari und G. Rozenberg, Hrsg. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific Publishing Co., Inc., 1999. ISBN: 978-981-02-4021-9 (siehe S. 42).

[EKT09] K. Ehrig, J. M. Küster und G. Taentzer. „Generating instance models from meta models“. In: *Software & Systems Modeling* 8.4 (2009), S. 479–500. ISSN: 1619-1366. DOI: 10.1007/s10270-008-0095-y (siehe S. 43, 130).

[EPS73] H. Ehrig, M. Pfender und H. J. Schneider. „Graph-grammars: An algebraic approach“. In: *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*. Okt. 1973, S. 167–180. DOI: 10.1109/SWAT.1973.11 (siehe S. 46).

[EPT04] H. Ehrig, U. Prange und G. Taentzer. „Fundamental Theory for Typed Attributed Graph Transformation“. In: *Graph Transformations*. Hrsg. von H. Ehrig, G. Engels, F. Parisi-Presicce und G. Rozenberg. Bd. 3256. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, S. 161–177. ISBN: 978-3-540-23207-0. DOI: 10.1007/978-3-540-30203-2_13 (siehe S. 48).

[Eri+12] A. Eriksson, B. Lindström, S. F. Andler und J. Offutt. „Model Transformation Impact on Test Artifacts: An Empirical Study“. In: *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*. MoDeVVa '12. Innsbruck, Österreich: ACM, 2012, S. 5–10. ISBN: 978-1-4503-1801-3. DOI: 10.1145/2427376.2427378 (siehe S. 190).

[Erm+10] C. Ermel, E. Biermann, J. Schmidt und A. Warning. „Visual Modeling of Controlled EMF Model Transformation using HENSHIN“. In: *ECEASST* 32 (2010). Hrsg. von J. de Lara und D. Varró. http://journal.ub.tu-berlin.de/eceasst/article/view/528. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 46).

[ES13] H. Ergin und E. Syriani. *Identification and Application of a Model Transformation Design Pattern*. ACM Southeast Conference, ACMSE'13. Apr. 2013. URL: http://syriani.cs.ua.edu/publications/ACMSE13.pdf (siehe S. 116).

[ES14a] H. Ergin und E. Syriani. *Implementations of Model Transformation Design Patterns Expressed in DelTa*. Techn. Ber. SERG-2014-01. http://software.eng.ua.edu/reports/SERG-2014-01.pdf. University of Alabama, USA, Feb. 2014 (siehe S. 116).

[ES14b] H. Ergin und E. Syriani. „Towards a Language for Graph-Based Model Transformation Design Patterns“. In: *Theory and Practice of Model Transformations*. Hrsg. von D. Di Ruscio und D. Varró. Bd. 8568. Lecture Notes in Computer Science. Springer International Publishing, 2014, S. 91–105. ISBN: 978-3-319-08788-7. DOI: 10.1007/978-3-319-08789-4_7 (siehe S. 116).

[Eva+99] A. Evans, R. France, K. Lano und B. Rumpe. „The UML as a Formal Modeling Notation“. In: *The Unified Modeling Language. «UML»'98: Beyond the Notation*. Hrsg. von J. Bézivin und P.-A. Muller. Bd. 1618. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, S. 336–348. ISBN: 978-3-540-66252-5. DOI: 10.1007/978-3-540-48480-6_26 (siehe S. 112).

[Eva98] A. S. Evans. „Reasoning with UML class diagrams“. In: *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*. 1998, S. 102–113. DOI: 10.1109/WIFT.1998.766304 (siehe S. 112).

[Fav04] J.-M. Favre. „Towards a Basic Theory to Model Model Driven Engineering“. In: *3rd Workshop in Software Model Engineering, WiSME*. 2004 (siehe S. 15, 20, 36).

[FD00] P. G. Frankl und Y. Deng. „Comparison of Delivered Reliability of Branch, Data Flow and Operational Testing: A Case Study“. In: *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '00. Portland, Oregon, USA: ACM, 2000, S. 124–134. ISBN: 1-58113-266-2. DOI: 10.1145/347324.348926 (siehe S. 5).

[FFP10] R. Ferreira, J. Faria und A. Paiva. „Test Coverage Analysis of UML Activity Diagrams for Interactive Systems“. In: *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*. Sep. 2010, S. 268–273. DOI: 10.1109/QUATIC.2010.51 (siehe S. 110).

[FI98] P. G. Frankl und O. Iakounenko. „Further Empirical Studies of Test Effectiveness“. In: *Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '98/FSE-6. Lake Buena Vista, Florida, USA: ACM, 1998, S. 153–162. ISBN: 1-58113-108-9. DOI: 10.1145/288195.288298 (siehe S. 5).

[Fis+00] T. Fischer, J. Niere, L. Torunski und A. Zündorf. „Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java“. In: *Theory and Application of Graph Transformations*. Hrsg. von H. Ehrig, G. Engels, H.-J. Kreowski und G. Rozenberg. Bd. 1764. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, S. 296–309. ISBN: 978-3-540-67203-6. DOI: 10.1007/978-3-540-46464-8_21 (siehe S. 53, 54, 60).

[FL08] U. Farooq und C. P. Lam. „Mutation Analysis for the Evaluation of AD Models“. In: *Computational Intelligence for Modelling Control Automation, 2008 International Conference on*. Dez. 2008, S. 296–301. DOI: 10.1109/CIMCA.2008.210 (siehe S. 203).

[Fle+09] F. Fleurey, B. Baudry, P.-A. Muller und Y. Le Traon. „Qualifying input test data for model transformations“. In: *Software & Systems Modeling* 8.2 (2009), S. 185–203. ISSN: 1619-1366. DOI: 10.1007/s10270-007-0074-8 (siehe S. 126, 127).

[FMM13] L. Fürst, M. Mernik und V. Mahnič. „Converting metamodels to graph grammars: doing without advanced graph grammar features“. In: *Software & Systems Modeling* (2013), S. 1–21. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0380-2 (siehe S. 130).

[FNT98] T. Fischer, J. Niere und L. Torunski. „Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling“. Diplomarbeit. Arbeitsgruppe Softwaretechnik, Warburger Straße 100, 33098 Paderborn: Universität-Gesamthochschule Paderborn, Juli 1998 (siehe S. 53).

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. mit Beiträgen von Kent Beck, John Brant, William Opdyke und Don Roberts. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2 (siehe S. 252).

[Fra+06] R. France, S. Ghosh, T. Dinh-Trong und A. Solberg. „Model-driven development using UML 2.0: promises and pitfalls“. In: *Computer* 39.2 (2006), S. 59–66. ISSN: 0018-9162. DOI: 10.1109/MC.2006.65 (siehe S. 15).

[FSB04] F. Fleurey, J. Steel und B. Baudry. „Validation in model-driven engineering: testing model transformations“. In: *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*. 2004, S. 29–40. DOI: 10.1109/MODEVA.2004.1425846 (siehe S. 5, 124, 125, 128, 130).

[Fug93] A. Fuggetta. „A classification of CASE technology“. In: *Computer* 26.12 (Dez. 1993), S. 25–38. ISSN: 0018-9162. DOI: 10.1109/2.247645 (siehe S. 333).

[FW07] S. Förtsch und B. Westfechtel. „Differencing and Merging of Software Diagrams - State of the Art and Challenges“. In: *ICSOFT 2007, Proceedings of the Second International Conference on Software and Data Technologies, Volume SE, Barcelona, Spain, July 22-25, 2007*. Hrsg. von J. Filipe, B. Shishkov und M. Helfert. INSTICC Press, 2007, S. 90–99. ISBN: 978-989-8111-06-7 (siehe S. 131, 132).

[FW88] P. G. Frankl und E. J. Weyuker. „An applicable family of data flow testing criteria“. In: *Software Engineering, IEEE Transactions on* 14.10 (Okt. 1988), S. 1483–1498. ISSN: 0098-5589. DOI: 10.1109/32.6194 (siehe S. 93, 94, 263, 287).

[FW93] P. G. Frankl und S. N. Weiss. „An experimental comparison of the effectiveness of branch testing and data flow testing“. In: *Software Engineering, IEEE Transactions on* 19.8 (Aug. 1993), S. 774–787. ISSN: 0098-5589. DOI: 10.1109/32.238581 (siehe S. 5, 87, 268).

[Gam+95] E. Gamma, R. Helm, R. Johnson und J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2 (siehe S. 37, 115, 116).

[GBD07] L. Geiger, T. Buchmann und A. Dotor. „EMF Code Generation with Fujaba“. In: *5th International Fujaba Days*. Kassel, Deutschland, 8-09. Oktober 2007 (siehe S. 54).

[GBR05] M. Gogolla, J. Bohling und M. Richters. „Validating UML and OCL models in USE by automatic snapshot generation“. In: *Software & Systems Modeling* 4.4 (2005), S. 386–398. ISSN: 1619-1366. DOI: 10.1007/s10270-005-0089-y (siehe S. 113, 127, 286).

[GC12] C. A. González und J. Cabot. „ATLTest: A White-Box Test Generation Approach for ATL Transformations“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von R. B. France, J. Kazmeier, R. Breu und C. Atkinson. Bd. 7590. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 449–464. ISBN: 978-3-642-33665-2. DOI: 10.1007/978-3-642-33666-9_29 (siehe S. 128, 138).

[Gei+06] R. Geiß, G. V. Batz, D. Grund, S. Hack und A. Szalkowski. „GrGen: A Fast SPO-Based Graph Rewriting Tool“. In: *Graph Transformations*. Hrsg. von A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro und G. Rozenberg. Bd. 4178. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 383–397. ISBN: 978-3-540-38870-8. DOI: 10.1007/11841883_27 (siehe S. 46).

[Gei11] L. Geiger. „Fehlersuche im Modell: Modellbasiertes Testen und Debuggen“. http://d-nb.info/101373873X. Diss. University of Kassel, 2011 (siehe S. 5, 135, 139, 140, 149, 190, 278).

[Ger+02] A. Gerber, M. Lawley, K. Raymond, J. Steel und A. Wood. „Transformation: The Missing Link of MDA“. In: *Graph Transformation*. Hrsg. von A. Corradini, H. Ehrig, H.-J. Kreowski und G. Rozenberg. Bd. 2505. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, S. 90–105. ISBN: 978-3-540-44310-0. DOI: 10.1007/3-540-45832-8_9 (siehe S. 30, 83).

[GG93] M. Grochtmann und K. Grimm. „Classification trees for partition testing“. In: *Software Testing, Verification and Reliability* 3.2 (1993), S. 63–82. ISSN: 1099-1689. DOI: 10.1002/stvr.4370030203 (siehe S. 126).

[Gho+03] S. Ghosh, R. France, C. Braganza, N. Kawane, A. Andrews und O. Pilskalns. „Test adequacy assessment for UML design model testing“. In: *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*. Nov. 2003, S. 332–343. DOI: 10.1109/ISSRE.2003.1251054 (siehe S. 108).

[GHS09] H. Giese, S. Hildebrandt und A. Seibel. „Improved Flexibility and Scalability by Interpreting Story Diagrams“. In: *ECEASST* 18 (2009). Hrsg. von A. Boronat und R. Heckel. http://journal.ub.tu-berlin.de/eceasst/article/view/268. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 53, 54).

[Gie+06] H. Giese, S. Glesner, J. Leitner, W. Schäfer und R. Wagner. „Towards Verified Model Transformations“. In: *MoDeV²a: Model Development, Validation and Verification (Proceedings 3rd International Workshop)*. Hrsg. von B. Baudry, D. Hearnden, N. Papin und J. G. Süß. http://modeva.itee.uq.edu.au/accepted_papers/main.pdf (zuletzt abgerufen am 25.9.2014). Oktober 2006, S. 78–93 (siehe S. 120).

[Gie+12] H. Giese, L. Lambers, B. Becker, S. Hildebrandt, S. Neumann, T. Vogel und S. Wätzoldt. „Graph Transformations for MDE, Adaptation, and Models at Runtime“. In: *Formal Methods for Model-Driven Engineering*. Hrsg. von M. Bernardo, V. Cortellessa und A. Pierantonio. Bd. 7320. Lecture Notes in

Computer Science. Springer Berlin Heidelberg, 2012, S. 137–191. ISBN: 978-3-642-30981-6. DOI: 10.1007/978-3-642-30982-3_5 (siehe S. 46, 50, 51, 54).

[GMH81] J. Gannon, P. McMullin und R. Hamlet. „Data Abstraction, Implementation, Specification, and Testing“. In: *ACM Transactions on Programming Languages and Systems* 3.3 (Juli 1981), S. 211–223. ISSN: 0164-0925. DOI: 10.1145/357139.357140 (siehe S. 106, 197).

[GOA05] M. Grindal, J. Offutt und S. F. Andler. „Combination testing strategies: a survey“. In: *Software Testing, Verification and Reliability* 15.3 (2005), S. 167–199. ISSN: 1099-1689. DOI: 10.1002/stvr.319 (siehe S. 97).

[Gon+12] C. A. Gonzalez, F. Buttner, R. Clarisó und J. Cabot. „EMFtoCSP: A tool for the lightweight verification of EMF models“. In: *Software Engineering: Rigorous and Agile Approaches (FormSERA), 2012 Formal Methods in*. Juni 2012, S. 44–50. DOI: 10.1109/FormSERA.2012.6229788 (siehe S. 113, 129, 286).

[GS13] E. Guerra und M. Soeken. „Specification-driven model transformation testing“. In: *Software & Systems Modeling* (2013), S. 1–22. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0369-x (siehe S. 129, 130, 135, 288).

[GSR05] L. Geiger, C. Schneider und C. Reckord. „Template- and modelbased code generation for MDA-Tools“. In: *3rd International Fujaba Days*. Paderborn, Deutschland, Sep. 2005 (siehe S. 54).

[Gue+13] E. Guerra, J. de Lara, M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck und W. Schwinger. „Automated verification of model transformations based on visual contracts“. In: *Automated Software Engineering* 20.1 (2013), S. 5–46. ISSN: 0928-8910. DOI: 10.1007/s10515-012-0102-y (siehe S. 106, 133–135).

[Gue12] E. Guerra. „Specification-Driven Test Generation for Model Transformations“. In: *Theory and Practice of Model Transformations*. Hrsg. von Z. Hu und J. de Lara. Bd. 7307. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 40–55. ISBN: 978-3-642-30475-0. DOI: 10.1007/978-3-642-30476-7_3 (siehe S. 130, 135).

[GV03] C. Girault und R. Valk. *Petri Nets for Systems Engineering - A Guide to Modeling, Verification, and Applications*. Springer Berlin Heidelberg, 2003, S. XVI, 607. ISBN: 978-3-642-07447-9. DOI: 10.1007/978-3-662-05324-9 (siehe S. 17).

[GV11] M. Gogolla und A. Vallecillo. „Tractable Model Transformation Testing“. In: *Modelling Foundations and Applications*. Hrsg. von R. B. France, J. M. Kuester, B. Bordbar und R. F. Paige. Bd. 6698. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 221–235. ISBN: 978-3-642-21469-1. DOI: 10.1007/978-3-642-21470-7_16 (siehe S. 127).

[GWZ94] A. Goldberg, T. C. Wang und D. Zimmerman. „Applications of Feasible Path Analysis to Program Testing“. In: *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '94. Seattle, Washington, USA: ACM, 1994, S. 80–94. ISBN: 0-89791-683-2. DOI: 10.1145/186258.186523 (siehe S. 263, 265).

[GZ05] L. Geiger und A. Zündorf. „Story Driven Testing - SDT“. In: *SIGSOFT Softw. Eng. Notes* 30.4 (Mai 2005), S. 1–6. ISSN: 0163-5948. DOI: 10.1145/1082983.1083186 (siehe S. 6, 133, 136).

[GZ06] L. Geiger und A. Zündorf. „Tool Modeling with Fujaba“. In: *Electronic Notes in Theoretical Computer Science* 148.1 (2006). Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004), S. 173–186. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.12.017 (siehe S. 54).

[Hal+91] N. Halbwachs, P. Caspi, P. Raymond und D. Pilaud. „The synchronous data flow programming language LUSTRE“. In: *Proceedings of the IEEE* 79.9 (Sep. 1991), S. 1305–1320. ISSN: 0018-9219. DOI: 10.1109/5.97300 (siehe S. 106).

[Hal90] A. Hall. „Seven myths of formal methods“. In: *Software, IEEE* 7.5 (Sep. 1990), S. 11–19. ISSN: 0740-7459. DOI: 10.1109/52.57887 (siehe S. 5).

[Har87] D. Harel. „Statecharts: a visual formalism for complex systems“. In: *Science of Computer Programming* 8.3 (1987), S. 231–274. ISSN: 0167-6423. DOI: 10.1016/0167-6423(87)90035-9 (siehe S. 19).

[Hec06] R. Heckel. „Graph Transformation in a Nutshell“. In: *Electronic Notes in Theoretical Computer Science* 148.1 (Feb. 2006), S. 187–198. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.12.018 (siehe S. 45).

[Hec98] R. Heckel. „Compositional verification of reactive systems specified by graph transformation“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von E. Astesiano. Bd. 1382. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, S. 138–153. ISBN: 978-3-540-64303-6. DOI: 10.1007/BFb0053588 (siehe S. 122).

[Hes06] W. Hesse. „More matters on (meta-)modelling: remarks on Thomas Kühne's ‚matters'“. In: *Software & Systems Modeling* 5.4 (2006), S. 387–394. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0033-9 (siehe S. 16).

[Hil+12] S. Hildebrandt, L. Lambers, H. Giese, D. Petrick und I. Richter. „Automatic Conformance Testing of Optimized Triple Graph Grammar Implementations“. In: *Applications of Graph Transformations with Industrial Relevance*. Hrsg. von A. Schürr, D. Varró und G. Varró. Bd. 7233. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 238–253. ISBN: 978-3-642-34175-5. DOI: 10.1007/978-3-642-34176-2_20 (siehe S. 7, 125, 131, 133, 136).

[HKM11] R. Heckel, T. A. Khan und R. Machado. „Towards Test Coverage Criteria for Visual Contracts“. In: *ECEASST* 41 (2011). Hrsg. von F. Gadducci und L. Mariani. http://journal.ub.tu-berlin.de/eceasst/article/view/667. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 5, 133, 136, 138, 287, 288).

[HKT02] R. Heckel, J. M. Küster und G. Taentzer. „Confluence of Typed Attributed Graph Transformation Systems“. In: *Graph Transformation*. Hrsg. von A. Corradini, H. Ehrig, H.-J. Kreowski und G. Rozenberg. Bd. 2505. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, S. 161–176. ISBN: 978-3-540-44310-0. DOI: 10.1007/3-540-45832-8_14 (siehe S. 121).

[HL03] R. Heckel und M. Lohmann. „Towards Model-Driven Testing“. In: *Electronic Notes in Theoretical Computer Science* 82.6 (2003). TACoS'03, International Workshop on Test and Analysis of Component-Based Systems (Satellite Event of ETAPS 2003), S. 33–43. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)81023-5 (siehe S. 6).

[HL05] R. Heckel und M. Lohmann. „Towards Contract-based Testing of Web Services“. In: *Electronic Notes in Theoretical Computer Science* 116 (2005). Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004) Test and Analysis of Component Based Systems 2004, S. 145–156. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2004.02.073 (siehe S. 6, 125).

[HL07] R. Heckel und M. Lohmann. „Model-driven development of reactive information systems: from graph transformation rules to JML contracts“. In: *International Journal on Software Tools for Technology Transfer* 9.2 (2007), S. 193–207. ISSN: 1433-2779. DOI: 10.1007/s10009-006-0020-z (siehe S. 6, 133, 136, 283).

[HLG13] S. Hildebrandt, L. Lambers und H. Giese. „Complete Specification Coverage in Automatically Generated Conformance Test Cases for TGG Implementations“. In: *Theory and Practice of Model Transformations*. Hrsg. von K. Duddy und G. Kappel. Bd. 7909. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 174–188. ISBN: 978-3-642-38882-8. DOI: 10.1007/978-3-642-38883-5_16 (siehe S. 131, 136).

[HM05] R. Heckel und L. Mariani. „Automatic Conformance Testing of Web Services“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von M. Cerioli. Bd. 3442. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 34–48. ISBN: 978-3-540-25420-1. DOI: 10.1007/978-3-540-31984-9_4 (siehe S. 6, 106, 125, 136).

[HM10] B. Hoffmann und M. Minas. „Defining Models – Meta Models versus Graph Grammars“. In: *ECEASST* 29 (2010). Hrsg. von J. Küster und E. Tuosto. http://journal.ub.tu-berlin.de/eceasst/article/view/411. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 130).

[HM11] B. Hoffmann und M. Minas. „Generating Instance Graphs from Class Diagrams with Adaptive Star Grammars“. In: *ECEASST* 39 (2011). Hrsg. von R. Echahed, A. Habel und M. Mosbah. http://journal.ub.tu-berlin.de/eceasst/article/view/650. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 130).

[Hoa69] C. A. R. Hoare. „An Axiomatic Basis for Computer Programming“. In: *Commun. ACM* 12.10 (Oktober 1969), S. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259 (siehe S. 123).

[Hof99] D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Anniversary Edition. Basic Books, Jan. 1999. ISBN: 978-0465026562 (siehe S. 1).

[Hol+11] A. Holzer, V. Januzaj, S. Kugele, B. Langer, C. Schallhart, M. Tautschnig und H. Veith. „Seamless Testing for Models and Code“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von D. Giannakopoulou und F. Orejas. Bd. 6603. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 278–293. ISBN: 978-3-642-19810-6. DOI: 10.1007/978-3-642-19811-3_20 (siehe S. 110).

[Hol97] G. J. Holzmann. „The model checker SPIN“. In: *Software Engineering, IEEE Transactions on* 23.5 (Mai 1997), S. 279–295. ISSN: 0098-5589. DOI: 10.1109/32.588521 (siehe S. 122).

[How76] W. E. Howden. „Reliability of the Path Analysis Testing Strategy“. In: *Software Engineering, IEEE Transactions on* SE-2.3 (Sep. 1976), S. 208–215. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233816 (siehe S. 89).

[How82] W. E. Howden. „Weak Mutation Testing and Completeness of Test Sets“. In: *Software Engineering, IEEE Transactions on* SE-8.4 (Juli 1982), S. 371–379. ISSN: 0098-5589. DOI: 10.1109/TSE.1982.235571 (siehe S. 98, 101).

[HR04] M. Huth und M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge, Vereinigtes Königreich: Cambridge University Press, 2004. ISBN: 978-0-521-54310-1 (siehe S. 94, 334).

[HT90] D. Hamlet und R. Taylor. „Partition Testing Does Not Inspire Confidence“. In: *Software Engineering, IEEE Transactions on* 16.12 (Dez. 1990), S. 1402–1411. ISSN: 0098-5589. DOI: 10.1109/32.62448 (siehe S. 5).

[HU72] M. S. Hecht und J. D. Ullman. „Flow Graph Reducibility“. In: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*. STOC '72. Denver, Colorado, USA: ACM, 1972, S. 238–250. DOI: 10.1145/800152.804919 (siehe S. 60, 207).

[HU74] M. S. Hecht und J. D. Ullman. „Characterizations of Reducible Flow Graphs“. In: *Journal of the ACM* 21.3 (Juli 1974), S. 367–375. ISSN: 0004-5411. DOI: 10.1145/321832.321835 (siehe S. 60, 207).

[Hut+94] M. Hutchins, H. Foster, T. Goradia und T. Ostrand. „Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria“. In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE '94. Sorrento, Italien: IEEE Computer Society Press, 1994, S. 191–200. ISBN: 0-8186-5855-X (siehe S. 5, 87).

[HW95] R. Heckel und A. Wagner. „Ensuring Consistency of Conditional Graph Grammars – A Constructive Approach“. In: *Electronic Notes in Theoretical Computer Science* 2 (1995). SEGRAGRA 1995, Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation, S. 118–126. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(05)80188-4 (siehe S. 123).

[IEC01] IEC (International Electrotechnical Commission). *Hazard and operability studies (HAZOP studies) – Application guide*. 2001 (siehe S. 333).

[IEE08] IEEE. „IEEE Standard for Software and System Test Documentation“. In: *IEEE Std 829-2008* (2008). DOI: 10.1109/IEEESTD.2008.4578383 (siehe S. 88).

[IEE90] IEEE. „IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Std 610.12-1990* (1990). ISBN 1-55937467-X, S. 1–84. DOI: 10.1109/IEEESTD.1990.101064 (siehe S. 79, 80, 84).

[IEE98] IEEE. „IEEE Standard for a Software Quality Metrics Methodology“. In: *IEEE Std 1061-1998* (Dez. 1998), S. i–v, 1–88. DOI: 10.1109/IEEESTD.1998.243394 (siehe S. 115).

[IH14] L. Inozemtseva und R. Holmes. „Coverage is Not Strongly Correlated with Test Suite Effectiveness“. In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, Indien: ACM, 2014, S. 435–445. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568271 (siehe S. 5, 87).

[Irv+07] S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis und M. Utting. „Jumble Java Byte Code to Measure the Effectiveness of Unit Tests“. In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*. Sep. 2007, S. 169–175. DOI: 10.1109/TAIC.PART.2007.38 (siehe S. 198).

[ISH08] M.-E. Iacob, M. W. A. Steen und L. Heerink. „Reusable Model Transformation Patterns“. In: *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*. Sep. 2008, S. 1–10. DOI: 10.1109/EDOCW.2008.51 (siehe S. 116).

[ITU11] International Telecommunication Union. (ITU). *Message Sequence Chart (MSC)*. http://www.itu.int/rec/T-REC-Z.120. Recommendation „ITU-T Z.120“. Feb. 2011 (siehe S. 19).

[Jac02] D. Jackson. „Alloy: A Lightweight Object Modelling Notation“. In: *ACM Transactions on Software Engineering and Methodology* 11.2 (Apr. 2002), S. 256–290. ISSN: 1049-331X. DOI: 10.1145/505145.505149 (siehe S. 112, 286).

[JBK10] E. Jakumeit, S. Buchwald und M. Kroll. „GrGen.NET“. In: *International Journal on Software Tools for Technology Transfer* 12.3-4 (2010), S. 263–271. ISSN: 1433-2779. DOI: 10.1007/s10009-010-0148-8 (siehe S. 46).

[Jec+04] M. Jeckle, C. Rupp, J. Hahn, B. Zengler und S. Queins. *UML 2 glasklar*. Carl Hanser Verlag, 2004. ISBN: 3-446-22575-7 (siehe S. 17).

[Jen81] K. Jensen. „Coloured petri nets and the invariant-method“. In: *Theoretical Computer Science* 14.3 (1981), S. 317–336. ISSN: 0304-3975. DOI: 10.1016/0304-3975(81)90049-9 (siehe S. 119).

[JH11] Y. Jia und M. Harman. „An Analysis and Survey of the Development of Mutation Testing“. In: *Software Engineering, IEEE Transactions on* 37.5 (Sep. 2011), S. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62 (siehe S. 98, 100).

[Joh+09] J. Johannes, S. Zschaler, M. A. Fernández, A. Castillo, D. S. Kolovos und R. F. Paige. „Abstracting Complex Languages through Transformation and Composition“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von A. Schürr und B. Selic. Bd. 5795. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 546–550. ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_41 (siehe S. 116).

[Jou+06] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev und P. Valduriez. „ATL: A QVT-like Transformation Language“. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, S. 719–720. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176691 (siehe S. 333).

[JSM14] A. K. Jena, S. K. Swain und D. P. Mohapatra. „A Novel Approach for Test Case Generation from UML Activity Diagram“. In: *Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on*. Feb. 2014, S. 621–629. DOI: 10.1109/ICICICT.2014.6781352 (siehe S. 111).

[KA07] J. M. Küster und M. Abd-El-Razik. „Validation of Model Transformations – First Experiences Using a White Box Approach“. In: *Models in Software Engineering*. Hrsg. von T. Kühne. Bd. 4364. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 193–204. ISBN: 978-3-540-69488-5. DOI: 10.1007/978-3-540-69489-2_24 (siehe S. 106, 127–129, 134, 136, 138, 204, 205).

[Kan+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak und A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Techn. Ber. CMU/SEI-90-TR-21. Carnegie-Mellon University Software Engineering Institute, Nov. 1990 (siehe S. 32).

[Kas74] T. Kasai. „Translatability of flowcharts into while programs“. In: *Journal of Computer and System Sciences* 9.2 (1974), S. 177–195. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(74)80006-1 (siehe S. 60).

[KBA02] I. Kurtev, J. Bézivin und M. Aksit. „Technological Spaces: An Initial Appraisal“. In: *Proceedings of the International Symposium on Distributed Objects and Applications, DOA 2002*. 2002 (siehe S. 21, 34).

[KCM00] S. Kim, J. A. Clark und J. A. McDermid. „Class Mutation: Mutation Testing for Object-Oriented Programs“. In: *Tagungsband der NET.ObjectDays (1. vereinigte GI Fachtagung „Objektorientierte Programmierung für die vernetzte Welt“)*. 2000 (siehe S. 201).

[Ken02] S. Kent. „Model Driven Engineering“. In: *Integrated Formal Methods*. Hrsg. von M. Butler, L. Petre und K. Sere. Bd. 2335. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, S. 286–298. ISBN: 978-3-540-43703-1. DOI: 10.1007/3-540-47884-1_16 (siehe S. 15).

[KH13] Y. Khan und J. Hassine. „Mutation Operators for the Atlas Transformation Language“. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*. März 2013, S. 43–52. DOI: 10.1109/ICSTW.2013.13 (siehe S. 129, 204).

[KHG11] M. Kuhlmann, L. Hamann und M. Gogolla. „Extensive Validation of OCL Models by Integrating SAT Solving into USE“. In: *Objects, Models, Components, Patterns*. Hrsg. von J. Bishop und A. Vallecillo. Bd. 6705. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 290–306. ISBN: 978-3-642-21951-1. DOI: 10.1007/978-3-642-21952-8_21 (siehe S. 113, 132).

[Kim+07] H. Kim, S. Kang, J. Baik und I. Ko. „Test Cases Generation from UML Activity Diagrams“. In: *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*. Bd. 3. Juli 2007, S. 556–561 (siehe S. 110, 111).

[Kim+99] Y. G. Kim, H. S. Hong, D.-H. Bae und S. D. Cha. „Test cases generation from UML state diagrams“. In: *IEE Proceedings Software* 146.4 (Aug. 1999), S. 187–192. ISSN: 1462-5970. DOI: 10.1049/ip-sen:19990602 (siehe S. 93, 94, 106).

[Kin76] J. C. King. „Symbolic Execution and Program Testing“. In: *Commun. ACM* 19.7 (Juli 1976), S. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252 (siehe S. 106).

[Kir09] R. Kirner. „Towards Preserving Model Coverage and Structural Code Coverage“. In: *EURASIP J. Embedded Syst.* 2009 (Jan. 2009), 6:1–6:16. ISSN: 1687-3955. DOI: 10.1155/2009/127945 (siehe S. 190).

[KK05] B. König und V. Kozioura. „AUGUR – A Tool for the Analysis of Graph Transformation Systems“. In: *Bulletin of the European Association for Theoretical Computer Science (EATCS)* 87 (2005). http://www.eatcs.org/images/bulletin/beatcs87.pdf (zuletzt abgerufen am 4.10.2014), S. 126–137. ISSN: 0252-9742 (siehe S. 121).

[KK08] B. König und V. Kozioura. „Augur 2 – A New Version of a Tool for the Analysis of Graph Transformation Systems“. In: *Electronic Notes in Theoretical Computer Science* 211 (2008). Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006), S. 201–210. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.042 (siehe S. 121).

[KLV05] P. Klint, R. Lämmel und C. Verhoef. „Toward an Engineering Discipline for Grammarware“. In: *ACM Transactions on Software Engineering and Methodology* 14.3 (Juli 2005), S. 331–380. ISSN: 1049-331X. DOI: 10.1145/1072997.1073000 (siehe S. 36).

[KN08] G. Karsai und A. Narayanan. „Towards Verification of Model Transformations Via Goal-Directed Certification“. In: *Model-Driven Development of Reliable Automotive Services*. Hrsg. von M. Broy, I. H. Krüger und M. Meisinger. Bd. 4922. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 67–83. ISBN: 978-3-540-70929-9. DOI: 10.1007/978-3-540-70930-5_5 (siehe S. 120).

[Kol+09] D. S. Kolovos, D. Di Ruscio, A. Pierantonio und R. F. Paige. „Different models for model matching: An analysis of approaches to support model differencing“. In: *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on*. Mai 2009, S. 1–6. DOI: 10.1109/CVSM.2009.5071714 (siehe S. 132).

[Kol+14] D. Kolovos, L. Rose, A. García-Domínguez und R. Paige. *The Epsilon Book*. https://www.eclipse.org/epsilon/doc/book/ (zuletzt abgerufen am 14.10.2014). Sep. 2014 (siehe S. 132).

[KP05] A. S. Kossatchev und M. A. Posypkin. „Survey of compiler testing methods“. In: *Programming and Computer Software* 31.1 (2005), S. 10–19. ISSN: 0361-7688. DOI: 10.1007/s11086-005-0008-6 (siehe S. 125, 133).

[KPP06] D. S. Kolovos, R. F. Paige und F. A. Polack. „Model Comparison: A Foundation for Model Composition and Model Transformation Testing“. In: *Proceedings of the 2006 International Workshop on Global Integrated Model Management*. GaMMa '06. Shanghai, China: ACM, 2006, S. 13–20. ISBN: 1-59593-410-3. DOI: 10.1145/1138304.1138308 (siehe S. 131, 132).

[Kre81] H.-J. Kreowski. „A comparison between petri-nets and graph grammars“. In: *Graphtheoretic Concepts in Computer Science*. Hrsg. von H. Noltemeier. Bd. 100. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1981, S. 306–317. ISBN: 978-3-540-10291-5. DOI: 10.1007/3-540-10291-4_22 (siehe S. 42).

[KRH12a] T. A. Khan, O. Runge und R. Heckel. „Testing against Visual Contracts: Model-Based Coverage“. In: *Graph Transformations*. Hrsg. von H. Ehrig, G. Engels, H.-J. Kreowski und G. Rozenberg. Bd. 7562. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 279–293. ISBN: 978-3-642-33653-9. DOI: 10.1007/978-3-642-33654-6_19 (siehe S. 5, 133, 138, 287).

[KRH12b] T. A. Khan, O. Runge und R. Heckel. „Visual Contracts as Test Oracle in AGG 2.0“. In: *ECEASST* 47 (2012). Hrsg. von A. Fish und L. Lambers. http://journal.ub.tu-berlin.de/eceasst/article/view/728. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 6, 106, 133).

[KS09] D. Kundu und D. Samanta. „A Novel Approach to Generate Test Cases from UML Activity Diagrams“. In: *Journal of Object Technology* 8.3 (Mai 2009), S. 65–83. ISSN: 1660-1769. DOI: 10.5381/jot.2009.8.3.a1 (siehe S. 111).

[Küh06] T. Kühne. „Matters of (Meta-) Modeling“. In: *Software & Systems Modeling* 5.4 (2006), S. 369–385. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0017-9 (siehe S. 16, 18, 20).

[Kus+13] A. Kusel, J. Etzlstorfer, E. Kapsammer, P. Langer, W. Retschitzegger, J. Schönböck, W. Schwinger und M. Wimmer. „A Survey on Incremental Model Transformation Approaches“. In: *Proceedings of Models and Evolution Workshop (ME) @MoDELS*. 2013 (siehe S. 32, 35).

[Küs06] J. M. Küster. „Definition and validation of model transformations“. In: *Software & Systems Modeling* 5.3 (2006), S. 233–259. ISSN: 1619-1366. DOI: 10.1007/s10270-006-0018-8 (siehe S. 4, 121).

[KWB03] A. G. Kleppe, J. Warmer und W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, Massachusetts, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0-321-19442-X (siehe S. 15, 29, 30, 38).

[KWN05] U. Kelter, J. Wehren und J. Niere. „A Generic Difference Algorithm for UML Models“. In: *Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen*. Hrsg. von P. Liggesmeyer, K. Pohl und M. Goedicke. Bd. 64. LNI. GI, 2005, S. 105–116. ISBN: 3-88579-393-8 (siehe S. 132).

[Lai02] R. Lai. „A survey of communication protocol testing“. In: *Journal of Systems and Software* 62.1 (2002), S. 21–46. ISSN: 0164-1212. DOI: 10.1016/S0164-1212(01)00132-7 (siehe S. 105).

[Lam00] A. v. Lamsweerde. „Formal Specification: A Roadmap“. In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE '00. Limerick, Irland: ACM, 2000, S. 147–159. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336546 (siehe S. 36, 106).

[Läm01] R. Lämmel. „Grammar Testing“. In: *Fundamental Approaches to Software Engineering*. Hrsg. von H. Hussmann. Bd. 2029. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, S. 201–216. ISBN: 978-3-540-41863-4. DOI: 10.1007/3-540-45314-8_15 (siehe S. 97, 136).

[LB93] M. Löwe und M. Beyer. „AGG - An Implementation of Algebraic Graph Rewriting“. In: *Rewriting Techniques and Applications, 5th International Conference, RTA-93, Montreal, Canada, June 16-18, 1993, Proceedings*. Hrsg. von C. Kirchner. Bd. 690. Lecture Notes in Computer Science. Springer-Verlag, 1993, S. 451–456 (siehe S. 46).

[LBJ06] Y. Le Traon, B. Baudry und J. Jezequel. „Design by Contract to Improve Software Vigilance“. In: *Software Engineering, IEEE Transactions on* 32.8 (Aug. 2006), S. 571–586. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.79 (siehe S. 133, 283).

[LBR99] G. T. Leavens, A. L. Baker und C. Ruby. „JML: A Notation for Detailed Design“. In: *Behavioral Specifications of Businesses and Systems*. Hrsg. von H. Kilov, B. Rumpe und I. Simmonds. Bd. 523. The Springer International Series in Engineering and Computer Science. Springer US, 1999, S. 175–188. ISBN: 978-1-4613-7383-4. DOI: 10.1007/978-1-4615-5229-1_12 (siehe S. 125, 334).

[LC08] K. Lano und D. Clark. „Model Transformation Specification and Verification“. In: *Quality Software, 2008. QSIC '08. The Eighth International Conference on*. Aug. 2008, S. 45–54. DOI: 10.1109/QSIC.2008.38 (siehe S. 31, 34, 118).

[Leb+14] E. Leblebici, A. Anjorin, A. Schürr, S. Hildebrandt, J. Rieke und J. Greenyer. „A Comparison of Incremental Triple Graph Grammar Tools“. In: *ECEASST* 67 (2014). Hrsg. von F. Hermann und S. Sauer. http://journal.ub.tu-berlin.de/eceasst/article/view/939. European Assoc. of Software Science and Technology (siehe S. 287).

[Léd+01] Á. Lédeczi, A. Bakay, M. Maróti, P. Völgyesi, G. Nordstrom, J. Sprinkle und G. Karsai. „Composing domain-specific design environments“. In: *Computer* 34.11 (Nov. 2001), S. 44–51. ISSN: 0018-9162. DOI: 10.1109/2.963443 (siehe S. 132).

[Leg13] E. Legros. „Definition of a Type System for Generic and Reflective Graph Transformations“. Dissertation. Techn. Univ. Darmstadt, 2013 (siehe S. 284).

[LG09] J. de Lara und E. Guerra. „Formal Support for QVT-Relations with Coloured Petri Nets“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von A. Schürr und B. Selic. Bd. 5795. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 256–270. ISBN: 978-3-642-04424-3. DOI: 10.1007/978-3-642-04425-0_19 (siehe S. 119).

[Lig02] P. Liggesmeyer. *Software-Qualität - Testen, Analysieren und Verifizieren von Software*. Spektrum Akad. Verl., 2002. ISBN: 3-8274-1118-1 (siehe S. 6, 79–83, 96, 99, 115, 133).

[Lig09] P. Liggesmeyer. *Software-Qualität - Testen, Analysieren und Verifizieren von Software (2. Aufl.)* Spektrum Akademischer Verlag, 2009. ISBN: 978-3-8274-2056-5. DOI: 10.1007/978-3-8274-2203-3 (siehe S. 4, 85, 97, 98, 102).

[LK11a] K. Lano und S. Kolahdouz-Rahimi. „Design Patterns for Model Transformations“. In: *ICSEA 2011, The Sixth International Conference on Software Engineering Advances*. 2011, S. 263–268 (siehe S. 3).

[LK11b] K. Lano und S. Kolahdouz-Rahimi. „Model-Driven Development of Model Transformations“. In: *Theory and Practice of Model Transformations*. Hrsg. von J. Cabot und E. Visser. Bd. 6707. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 47–61. ISBN: 978-3-642-21731-9. DOI: 10.1007/978-3-642-21732-6_4 (siehe S. 116).

[LK14] K. Lano und S. Kolahdouz-Rahimi. „Model-transformation Design Patterns“. In: *Software Engineering, IEEE Transactions on* 40.12 (Dez. 2014), S. 1224–1259. ISSN: 0098-5589. DOI: 10.1109/TSE.2014.2354344 (siehe S. 117).

[LKC12] K. Lano, S. Kolahdouz-Rahimi und T. Clark. „Comparing Verification Techniques for Model Transformations“. In: *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*. MoDeVVa '12. Innsbruck, Österreich: ACM, 2012, S. 23–28. ISBN: 978-1-4503-1801-3. DOI: 10.1145/2427376.2427381 (siehe S. 4, 118).

[LL05] H. Li und C. Lam. „Using Anti-Ant-like Agents to Generate Test Threads from the UML Diagrams“. In: *Testing of Communicating Systems*. Hrsg. von F. Khendek und R. Dssouli. Bd. 3502. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 69–80. ISBN: 978-3-540-26054-7. DOI: 10.1007/11430230_6 (siehe S. 111).

[LMK14] P. Langer, T. Mayerhofer und G. Kappel. „Semantic Model Differencing Utilizing Behavioral Semantics Specifications“. In: *Model-Driven Engineering Languages and Systems*. Hrsg. von J. Dingel, W. Schulte, I. Ramos, S. Abrahão und E. Insfran. Bd. 8767. Lecture Notes in Computer Science. Springer International Publishing, 2014, S. 116–132. ISBN: 978-3-319-11652-5. DOI: 10.1007/978-3-319-11653-2_8 (siehe S. 132).

[Löw93] M. Löwe. „Algebraic approach to single-pushout graph transformation“. In: *Theoretical Computer Science* 109.1–2 (1993), S. 181–224. ISSN: 0304-3975. DOI: 10.1016/0304-3975(93)90068-5 (siehe S. 46).

[LS05] L. Lúcio und M. Samer. „12 Technology of Test-Case Generation“. In: *Model-Based Testing of Reactive Systems*. Hrsg. von M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker und A. Pretschner. Bd. 3472. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, S. 323–354. ISBN: 978-3-540-26278-7. DOI: 10.1007/11498490_15 (siehe S. 88).

[LS06] R. Lämmel und W. Schulte. „Controllable Combinatorial Coverage in Grammar-Based Testing“. In: *Testing of Communicating Systems*. Hrsg. von M. Uyar, A. Duale und M. Fecko. Bd. 3964. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 19–38. ISBN: 978-3-540-34184-0. DOI: 10.1007/11754008_2 (siehe S. 136).

[Lud03] J. Ludewig. „Models in software engineering – an introduction“. In: *Software & Systems Modeling* 2.1 (2003), S. 5–14. ISSN: 1619-1366. DOI: 10.1007/s10270-003-0020-3 (siehe S. 15, 16).

[LY96] D. Lee und M. Yannakakis. „Principles and Methods of Testing Finite State Machines – A Survey“. In: *Proceedings of the IEEE* 84.8 (Aug. 1996), S. 1090–1123. ISSN: 0018-9219. DOI: 10.1109/5.533956 (siehe S. 105).

[LZG04] Y. Lin, J. Zhang und J. Gray. „Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development“. In: *Proc. Workshop on Best Practices for Model-Driven Software Development (at OOPSLA 2004)*. http://gray.cs.ua.edu/pubs/

oopsla-2004-mdsd.pdf (zuletzt abgerufen am 7.10.2014). Okt. 2004 (siehe S. 131).

[LZG05] Y. Lin, J. Zhang und J. Gray. „A Testing Framework for Model Transformations“. In: *Model-Driven Software Development*. Hrsg. von S. Beydeda, M. Book und V. Gruhn. Springer Berlin Heidelberg, 2005, S. 219–236. ISBN: 978-3-540-25613-7. DOI: 10.1007/3-540-28554-7_10 (siehe S. 5, 10, 131).

[Mau90] P. M. Maurer. „Generating test data with enhanced context-free grammars“. In: *Software, IEEE* 7.4 (Juli 1990), S. 50–55. ISSN: 0740-7459. DOI: 10.1109/52.56422 (siehe S. 136).

[MB07] A. Maraee und M. Balaban. „Efficient Reasoning About Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets“. In: *Model Driven Architecture- Foundations and Applications*. Hrsg. von D. H. Akehurst, R. Vogel und R. F. Paige. Bd. 4530. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 17–31. ISBN: 978-3-540-72900-6. DOI: 10.1007/978-3-540-72901-3_2 (siehe S. 112).

[MBL08] J.-M. Mottu, B. Baudry und Y. Le Traon. „Model transformation testing: oracle issue“. In: *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*. 2008, S. 105–112. DOI: 10.1109/ICSTW.2008.27 (siehe S. 90, 131).

[MBT06a] J.-M. Mottu, B. Baudry und Y. Traon. „Mutation Analysis Testing for Model Transformations“. In: *Model Driven Architecture – Foundations and Applications*. Hrsg. von A. Rensink und J. Warmer. Bd. 4066. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 376–390. ISBN: 978-3-540-35909-8. DOI: 10.1007/11787044_28 (siehe S. 5, 129, 198, 201, 203–205).

[MBT06b] J.-M. Mottu, B. Baudry und Y. Traon. „Reusable MDA Components: A Testing-for-Trust Approach“. In: *Model Driven Engineering Languages and Systems*. Hrsg. von O. Nierstrasz, J. Whittle, D. Harel und G. Reggio. Bd. 4199. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 589–603. ISBN: 978-3-540-45772-5. DOI: 10.1007/11880240_41 (siehe S. 129, 133).

[MC07] M. J. Mcgill und B. H. C. Cheng. *Test-Driven Development of a Model Transformation with Jemtte*. Techn. Ber. Software Engineering und Network Systems Laboratory, Michigan State University, USA, 2007 (siehe S. 283).

[McC76] T. J. McCabe. „A Complexity Measure“. In: *Software Engineering, IEEE Transactions on* SE-2.4 (Dez. 1976), S. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837 (siehe S. 115).

[MCV05] T. Mens, K. Czarnecki und P. Van Gorp. „04101 Discussion – A Taxonomy of Model Transformations“. In: *Language Engineering for Model-Driven Software Development*. Hrsg. von J. Bezivin und R. Heckel. Dagstuhl Seminar Proceedings 04101. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Deutschland, 2005 (siehe S. 31, 32).

[MD08a] P. Mohagheghi und V. Dehlen. „Developing a Quality Framework for Model-Driven Engineering“. In: *Models in Software Engineering*. Hrsg. von H. Giese. Bd. 5002. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 275–286. ISBN: 978-3-540-69069-6. DOI: 10.1007/978-3-540-69073-3_29 (siehe S. 114).

[MD08b] P. Mohagheghi und V. Dehlen. „Where Is the Proof? - A Review of Experiences from Applying MDE in Industry“. In: *Model Driven Architecture – Foundations and Applications*. Hrsg. von I. Schieferdecker und A. Hartman. Bd. 5095. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 432–443. ISBN: 978-3-540-69095-5. DOI: 10.1007/978-3-540-69100-6_31 (siehe S. 3).

[Mey92] B. Meyer. „Applying „Design by Contract““. In: *Computer* 25.10 (Okt. 1992), S. 40–51. ISSN: 0018-9162. DOI: 10.1109/2.161279 (siehe S. 125, 126).

[Mij+13] S. Mijatov, P. Langer, T. Mayerhofer und G. Kappel. „A Framework for Testing UML Activities Based on fUML“. In: *MoDeVVa@MoDELS*. Hrsg. von F. Boulanger, M. Famelis und D. Ratiu. Bd. 1069. CEUR Workshop Proceedings. CEUR-WS.org, 2013, S. 1–10 (siehe S. 110).

[Moh+09] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche und W. Gilani. „MDE Adoption in Industry: Challenges and Success Criteria“. In: *Models in Software Engineering*. Hrsg. von M. R. V. Chaudron. Bd. 5421. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 54–59. ISBN: 978-3-642-01647-9. DOI: 10.1007/978-3-642-01648-6_6 (siehe S. 2).

[MOK05] Y.-S. Ma, J. Offutt und Y. R. Kwon. „MuJava: an automated class mutation system“. In: *Software Testing, Verification and Reliability* 15.2 (2005), S. 97–133. ISSN: 1099-1689. DOI: 10.1002/stvr.308 (siehe S. 198, 201).

[Moo01] I. Moore. „Jester – A JUnit Test Tester“. In: *Proc. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP)*. 2001 (siehe S. 198).

[Mot+12] J.-M. Mottu, S. Sen, M. Tisi und J. Cabot. „Static Analysis of Model Transformations for Effective Test Generation“. In: *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. 2012, S. 291–300. DOI: 10.1109/ISSRE.2012.7 (siehe S. 129).

[MP05] J. A. McQuillan und J. F. Power. *A Survey of UML-Based Coverage Criteria for Software Testing*. Techn. Ber. NUIM-CS-TR-2005-08. National University of Ireland, 2005 (siehe S. 108, 125).

[MP09] J. A. McQuillan und J. F. Power. „White-Box Coverage Criteria for Model Transformations“. In: *Preliminary Proceedings, 1st International Workshop, MtATL 2009*. Hrsg. von F. Jouault. AtlanMod INRIA & EMN. 2009, S. 63–77 (siehe S. 128, 138).

[MRR11a] S. Maoz, J. O. Ringert und B. Rumpe. „A Manifesto for Semantic Model Differencing“. In: *Models in Software Engineering*. Hrsg. von J. Dingel und A. Solberg. Bd. 6627. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 194–203. ISBN: 978-3-642-21209-3. DOI: 10.1007/978-3-642-21210-9_19 (siehe S. 132).

[MRR11b] S. Maoz, J. O. Ringert und B. Rumpe. „ADDiff: Semantic Differencing for Activity Diagrams“. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szegedin, Ungarn: ACM, 2011, S. 179–189. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025140 (siehe S. 111).

[MRR11c] S. Maoz, J. O. Ringert und B. Rumpe. „CDDiff: Semantic Differencing for Class Diagrams“. In: *ECOOP 2011 – Object-Oriented Programming*. Hrsg. von M. Mezini. Bd. 6813. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, S. 230–254. ISBN: 978-3-642-22654-0. DOI: 10.1007/978-3-642-22655-7_12 (siehe S. 110, 132).

[MV06] T. Mens und P. Van Gorp. „A Taxonomy of Model Transformation“. In: *Electronic Notes in Theoretical Computer Science* 152 (2006), S. 125–142. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.10.021 (siehe S. 31, 32).

[Nag87] M. Nagl. „Set theoretic approaches to graph grammars“. In: *Graph-Grammars and Their Application to Computer Science*. Hrsg. von H. Ehrig, M. Nagl, G. Rozenberg und A. Rosenfeld. Bd. 291. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1987, S. 41–54. ISBN: 978-3-540-18771-4. DOI: 10.1007/3-540-18771-5_43 (siehe S. 47).

[NK08] A. Narayanan und G. Karsai. „Towards Verifying Model Transformations“. In: *Electronic Notes in Theoretical Computer Science* 211 (Apr. 2008), S. 191–200. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.041 (siehe S. 120).

[NNZ00] U. Nickel, J. Niere und A. Zündorf. „The FUJABA Environment“. In: *Proceedings of the 22nd International Conference on Software Engineering*. ICSE '00. Limerick, Irland: ACM, 2000, S. 742–745. ISBN: 1-58113-206-9. DOI: 10.1145/337180.337620 (siehe S. 53, 207).

[NS11] A. Nayak und D. Samanta. „Synthesis of test scenarios using UML activity diagrams“. In: *Software & Systems Modeling* 10.1 (2011), S. 63–89. ISSN: 1619-1366. DOI: 10.1007/s10270-009-0133-4 (siehe S. 110, 111).

[NWP02] T. Nipkow, M. Wenzel und L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Bd. 2283. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, S. XIV, 226. ISBN: 978-3-540-43376-7. DOI: 10.1007/3-540-45949-9 (siehe S. 113).

[NZJ13] U. Norbisrath, A. Zündorf und R. Jubeh. *Story Driven Modeling*. Self-Publishing-Plattform „CreateSpace“, Apr. 2013, S. 1–333. ISBN: 978-1-483-94925-3 (siehe S. 53).

[OA99] J. Offutt und A. Abdurazik. „Generating Tests from UML Specifications“. In: *«UML»'99 — The Unified Modeling Language*. Hrsg. von R. France und B. Rumpe. Bd. 1723. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, S. 416–429. ISBN: 978-3-540-66712-4. DOI: 10.1007/3-540-46852-8_30 (siehe S. 107, 202).

[OB88] T. J. Ostrand und M. J. Balcer. „The Category-partition Method for Specifying and Generating Fuctional Tests“. In: *Commun. ACM* 31.6 (Juni 1988), S. 676–686. ISSN: 0001-0782. DOI: 10.1145/62959.62964 (siehe S. 96, 109, 125).

[OC94] A. J. Offutt und W. M. Craft. „Using compiler optimization techniques to detect equivalent mutants“. In: *Software Testing, Verification and Reliability* 4.3 (1994), S. 131–154. ISSN: 1099-1689. DOI: 10.1002/stvr.4370040303 (siehe S. 99).

[Off+03] J. Offutt, S. Liu, A. Abdurazik und P. Ammann. „Generating test data from state-based specifications“. In: *Software Testing, Verification and Reliability* 13.1 (2003), S. 25–53. ISSN: 1099-1689. DOI: 10.1002/stvr.264 (siehe S. 93, 107).

[Off+96] A. J. Offutt, J. Pan, K. Tewary und T. Zhang. „An Experimental Evaluation of Data Flow and Mutation Testing“. In: *Software – Practice & Experience* 26.2 (Feb. 1996), S. 165–176. ISSN: 0038-0644. DOI: 10.1002/(SICI)1097-024X(199602)26:2<165::AID-SPE5>3.0.CO;2-K (siehe S. 5, 102).

[OMG11] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1*. http://www.omg.org/spec/QVT/1.1/PDF/. Dokument „formal/2011-01-01“. Jan. 2011 (siehe S. 29, 38, 335).

[OMG97] OMG. *Meta Object Facility (MOF) Specification: Joint Revised Submission*. Version 1.1, Dokument „ad/97-08-14“, http://www.omg.org/cgi-bin/doc?ad/97-08-14.pdf (zuletzt abgerufen am 20.10.2014). Sep. 1997 (siehe S. 20, 21).

[OP96] A. J. Offutt und J. Pan. „Detecting equivalent mutants and the feasible path problem“. In: *Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security. Proceedings of the Eleventh Annual Conference on*. Juni 1996, S. 224–236. DOI: 10.1109/CMPASS.1996.507890 (siehe S. 99).

[OWK03] D. Ohst, M. Welle und U. Kelter. „Differences Between Versions of UML Diagrams“. In: *SIGSOFT Softw. Eng. Notes* 28.5 (Sep. 2003), S. 227–236. ISSN: 0163-5948. DOI: 10.1145/949952.940102 (siehe S. 132).

[Pac+07] C. Pacheco, S. K. Lahiri, M. D. Ernst und T. Ball. „Feedback-Directed Random Test Generation“. In: *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. Mai 2007, S. 75–84. DOI: 10.1109/ICSE.2007.37 (siehe S. 285).

[Par69] B. Pareigis. *Kategorien und Funktoren*. Stuttgart: Vieweg+Teubner Verlag, 1969. ISBN: 978-3-519-02210-7. DOI: 10.1007/978-3-663-12190-9 (siehe S. 46).

[Pat+10] L. Patzina, S. Patzina, T. Piper und A. Schürr. „Monitor Petri Nets for Security Monitoring“. In: *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*. S&D4RCES '10. Wien, Österreich: ACM, 2010, 3:1–3:6. ISBN: 978-1-4503-0368-2. DOI: 10.1145/1868433.1868438 (siehe S. 246, 247, 250).

[Pat+13] L. Patzina, S. Patzina, T. Piper und P. Manns. „Model-Based Generation of Run-Time Monitors for AUTOSAR“. In: *Modelling Foundations and Applications*. Hrsg. von P. Van Gorp, T. Ritter und L. M. Rose. Bd. 7949. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 70–85. ISBN: 978-3-642-39012-8. DOI: 10.1007/978-3-642-39013-5_6 (siehe S. 247).

[Pat14a] L. Patzina. „Generierung von effizienten Security-/Safety-Monitoren aus modellbasierten Beschreibungen“. http://tuprints.ulb.tu-darmstadt.de/id/eprint/4133. Diss. Technische Universität Darmstadt, 2014 (siehe S. 246, 247).

[Pat14b] S. Patzina. „Entwicklung einer Spezifikationssprache zur modellbasierten Generierung von Security-/Safety-Monitoren zur Absicherung von (Eingebetteten) Systemen“. http://tuprints.ulb.tu-darmstadt.de/id/eprint/4132. Diss. Technische Universität Darmstadt, 2014 (siehe S. 246, 247).

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Bd. 828. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1994, S. XIX, 329. ISBN: 978-3-540-58244-1. DOI: 10.1007/BFb0030541 (siehe S. 113, 122, 123).

[Pet62] C. A. Petri. „Kommunikation mit Automaten“. Dissertation. Institut für Angewandte Mathematik der Universität Bonn, Wegelerstr. 10: Technische Hochschule Darmstadt, 1962 (siehe S. 17).

[Pet76] C. A. Petri. „General Net Theory“. In: *Proc. of the Joint IBM University of Newcastle upon Tyne Seminar*. Hrsg. von B. Shaw. http://www.informatik.uni-hamburg.de/TGI/mitarbeiter/profs/petri/doc/GeneralNetTheory.pdf (zuletzt abgerufen am 1.10.2014). University of Newcastle upon Tyne, Sep. 1976, S. 131–169 (siehe S. 17).

[Pil14] S. M. Piller. „Erweiterung eines Mutationsrahmenwerks für programmierte Graphtransformationen am Beispiel von SDM“. ES-S-0097 (Betreuer: Martin Wieber). Studienarbeit. Technische Universität Darmstadt, Fachgebiet Echtzeitsysteme, Feb. 2014 (siehe S. 219, 220, 284).

[Plu09] D. Plump. „The Graph Programming Language GP“. In: *Algebraic Informatics*. Hrsg. von S. Bozapalidis und G. Rahonis. Bd. 5725. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 99–122. ISBN: 978-3-642-03563-0. DOI: 10.1007/978-3-642-03564-7_6 (siehe S. 123).

[Plu93] D. Plump. „Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence“. In: *Term Graph Rewriting*. Hrsg. von M. R. Sleep, M. J. Plasmeijer und M. C. J. D. van Eekelen. Chichester, UK: John Wiley und Sons Ltd., 1993. Kap. 15, S. 201–213. ISBN: 0-471-93567-0 (siehe S. 121).

[Plu98] D. Plump. „Termination of Graph Rewriting is Undecidable“. In: *Fundam. Inf.* 33.2 (Feb. 1998), S. 201–209. ISSN: 0169-2968 (siehe S. 121).

[Poe08] I. Poernomo. „Proofs-as-Model-Transformations“. In: *Theory and Practice of Model Transformations*. Hrsg. von A. Vallecillo, J. Gray und A. Pierantonio. Bd. 5063. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 214–228. ISBN: 978-3-540-69926-2. DOI: 10.1007/978-3-540-69927-9_15 (siehe S. 120).

[PP10] C. M. Poskitt und D. Plump. „A Hoare Calculus for Graph Programs“. In: *Graph Transformations*. Hrsg. von H. Ehrig, A. Rensink, G. Rozenberg und A. Schürr. Bd. 6372. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 139–154. ISBN: 978-3-642-15927-5. DOI: 10.1007/978-3-642-15928-2_10 (siehe S. 123).

[PP12] C. M. Poskitt und D. Plump. „Hoare-Style Verification of Graph Programs“. In: *Fundam. Inf.* 118.1-2 (Jan. 2012), S. 135–175. ISSN: 0169-2968 (siehe S. 123).

[PPS11] S. Patzina, L. Patzina und A. Schürr. „Extending LSCs for Behavioral Signature Modeling“. In: *Future Challenges in Security and Privacy for Academia and Industry*. Hrsg. von J. Camenisch, S. Fischer-Hübner, Y. Murayama, A. Portmann und C. Rieder. Bd. 354. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2011, S. 293–304. ISBN: 978-3-642-21423-3. DOI: 10.1007/978-3-642-21424-0_24 (siehe S. 247).

[PR69] J. L. Pfaltz und A. Rosenfeld. „Web Grammars“. In: *IJCAI*. Hrsg. von D. E. Walker und L. M. Norton. erschienen in [WN69]. William Kaufmann, 1969, S. 609–620. ISBN: 0-934613-21-4 (siehe S. 42).

[Pri+10] C. Priesterjahn, M. Tichy, S. Henkler, M. Hirsch und W. Schäfer. „12 Fujaba4Eclipse Real-Time Tool Suite“. In: *Model-Based Engineering of Embedded Real-Time Systems*. Hrsg. von H. Giese, G. Karsai, E. Lee, B. Rumpe und B. Schätz. Bd. 6100. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 309–315. ISBN: 978-3-642-16276-3. DOI: 10.1007/978-3-642-16277-0_12 (siehe S. 53).

[PS94] B.-U. Pagel und H.-W. Six. *Software Engineering – Band 1: Die Phasen der Softwareentwicklung*. Bd. 1. Addison Wesley, 1994. ISBN: 3-89319-735-4 (siehe S. 6).

[Pur72] P. Purdom. „A sentence generator for testing parsers“. In: *BIT Numerical Mathematics* 12.3 (1972), S. 366–375. ISSN: 0006-3835. DOI: 10.1007/BF01932308 (siehe S. 97, 136).

[Rad00] A. Radermacher. „Support for Design Patterns through Graph Transformation Tools“. In: *Applications of Graph Transformations with Industrial Relevance*. Hrsg. von M. Nagl, A. Schürr und M. Münch. Bd. 1779. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, S. 111–126. ISBN: 978-3-540-67658-4. DOI: 10.1007/3-540-45104-8_9 (siehe S. 25).

[RBJ00] V. Rusu, L. du Bousquet und T. Jéron. „An Approach to Symbolic Test Generation“. In: *Integrated Formal Methods*. Hrsg. von W. Grieskamp, T. Santen und B. Stoddart. Bd. 1945. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, S. 338–357. ISBN: 978-3-540-41196-3. DOI: 10.1007/3-540-40911-4_20 (siehe S. 106).

[RD14] W. Reisig und J. Desel. „Konzepte der Petrinetze“. In: *Informatik-Spektrum* 37.3 (2014), S. 172–190. ISSN: 0170-6012. DOI: 10.1007/s00287-013-0758-0 (siehe S. 2).

[Ren03] A. Rensink. „Towards Model Checking Graph Grammars“. In: *Workshop on Automated Verification of Critical Systems (AVoCS)*. Hrsg. von M. Leuschel, S. Gruner und S. Lo Presti. als Techn. Ber. DSSE-TR-2003-2. University of Southampton, 2003, S. 150–160 (siehe S. 122).

[Ren04] A. Rensink. „The GROOVE Simulator: A Tool for State Space Generation“. In: *Applications of Graph Transformations with Industrial Relevance*. Hrsg. von J. L. Pfaltz, M. Nagl und B. Böhlen. Bd. 3062. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, S. 479–485. ISBN: 978-3-540-22120-3. DOI: 10.1007/978-3-540-25959-6_40 (siehe S. 46, 122).

[Ren10] A. Rensink. „The Edge of Graph Transformation – Graphs for Behavioural Specification“. In: *Graph Transformations and Model-Driven Engineering*. Hrsg. von G. Engels, C. Lewerentz, W. Schäfer, A. Schürr und B. Westfechtel. Bd. 5765. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 6–32. ISBN: 978-3-642-17321-9. DOI: 10.1007/978-3-642-17322-6_2 (siehe S. 42, 45, 46).

[REP12] D. Ruscio, R. Eramo und A. Pierantonio. „Model Transformations“. In: *Formal Methods for Model-Driven Engineering*. Hrsg. von M. Bernardo, V. Cortellessa und A. Pierantonio. Bd. 7320. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 91–136. ISBN: 978-3-642-30981-6. DOI: 10.1007/978-3-642-30982-3_4 (siehe S. 31, 32, 38).

[ROT89] D. Richardson, O. O'Malley und C. Tittle. „Approaches to Specification-based Testing“. In: *SIGSOFT Softw. Eng. Notes* 14.8 (Nov. 1989), S. 86–96. ISSN: 0163-5948. DOI: 10.1145/75309.75319 (siehe S. 125).

[Roz97] G. Rozenberg, Hrsg. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations*. River Edge, New Jersey, USA: World Scientific Publishing Co., Inc., 1997. ISBN: 98-102288-48 (siehe S. 41–46, 51).

[RS97] J. Rekers und A. Schürr. „Defining and Parsing Visual Languages with Layered Graph Grammars“. In: *Journal of Visual Languages & Computing* 8.1 (1997), S. 27–55. ISSN: 1045-926X. DOI: 10.1006/jvlc.1996.0027 (siehe S. 121, 130).

[RSV04] A. Rensink, Á. Schmidt und D. Varró. „Model Checking Graph Transformations: A Comparison of Two Approaches“. In: *Graph Transformations*. Hrsg. von H. Ehrig, G. Engels, F. Parisi-Presicce und G. Rozenberg. Bd. 3256. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, S. 226–241. ISBN: 978-3-540-23207-0. DOI: 10.1007/978-3-540-30203-2_17 (siehe S. 122).

[RW85] S. Rapps und E. J. Weyuker. „Selecting Software Test Data Using Data Flow Information“. In: *Software Engineering, IEEE Transactions on* SE-11.4 (Apr. 1985), S. 367–375. ISSN: 0098-5589. DOI: 10.1109/TSE.1985.232226 (siehe S. 93, 94).

[SAB10] S. M. A. Shah, K. Anastasakis und B. Bordbar. „From UML to Alloy and Back Again“. In: *Models in Software Engineering*. Hrsg. von S. Ghosh. Bd. 6002. Lecture Notes in Computer Science. Erweiterte Version. Springer Berlin Heidelberg, 2010, S. 158–171. ISBN: 978-3-642-12260-6. DOI: 10.1007/978-3-642-12261-3_16 (siehe S. 112).

[SBM09] S. Sen, B. Baudry und J.-M. Mottu. „Automatic Model Generation Strategies for Model Transformation Testing“. In: *Theory and Practice of Model Transformations*. Hrsg. von R. F. Paige. Bd. 5563. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 148–164. ISBN: 978-3-642-02407-8. DOI: 10.1007/978-3-642-02408-5_11 (siehe S. 130).

[SC03] I. Stürmer und M. Conrad. „Test suite design for code generation tools“. In: *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*. 2003, S. 286–290. DOI: 10.1109/ASE.2003.1240322 (siehe S. 125, 126, 133, 136).

[SC05] I. Stürmer und M. Conrad. „Ein Testverfahren für optimierende Codegeneratoren“. In: *Informatik - Forschung und Entwicklung* 19.4 (2005), S. 213–223. ISSN: 0178-3564. DOI: 10.1007/s00450-005-0189-5 (siehe S. 126, 133).

[SCD12] G. M. K. Selim, J. R. Cordy und J. Dingel. „Model Transformation Testing: The State of the Art“. In: *Proceedings of the First Workshop on the Analysis of Model Transformations*. AMT '12. Innsbruck, Österreich: ACM, 2012, S. 21–26. ISBN: 978-1-4503-1803-7. DOI: 10.1145/2432497.2432502 (siehe S. 124).

[Sch+07] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux und Y. Bontemps. „Generic semantics of feature diagrams“. In: *Computer Networks* 51.2 (2007), S. 456–479. ISSN: 1389-1286. DOI: 10.1016/j.comnet.2006.08.008 (siehe S. 32).

[Sch+13] J. Schönböck, G. Kappel, M. Wimmer, A. Kusel, W. Retschitzegger und W. Schwinger. „TetraBox - A Generic White-Box Testing Framework for Model Transformations“. In: *Proceedings of the 20th Asia-Pacific Software Engineering Conference (APSEC)*. 2013 (siehe S. 90, 129, 138).

[Sch06] D. Schmidt. „Guest Editor's Introduction: Model-Driven Engineering“. In: *Computer* 39.2 (2006), S. 25–31. ISSN: 0018-9162. DOI: 10.1109/MC.2006.58 (siehe S. 15).

[Sch91a] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen: formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung*. DUV: Informatik. D82 (Diss. RWTH Aachen), Herausgegeben und eingeleitet von Manfred Nagl. Deutscher Universitäts-Verlag (Springer Fachm. Wiesbaden), 1991, S. I–XIII, 1–461. ISBN: 978-3-8244-2021-6 (siehe S. 47, 52).

[Sch91b] A. Schürr. „Progress: A VHL-language based on graph grammars“. In: *Graph Grammars and Their Application to Computer Science*. Hrsg. von H. Ehrig, H.-J. Kreowski und G. Rozenberg. Bd. 532. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, S. 641–659. ISBN: 978-3-540-54478-4. DOI: 10.1007/BFb0017419 (siehe S. 47, 52).

[Sch95] A. Schürr. „Specification of graph translators with triple graph grammars“. In: *Graph-Theoretic Concepts in Computer Science*. Hrsg. von E. Mayr, G. Schmidt und G. Tinhofer. Bd. 903. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, S. 151–163. ISBN: 978-3-540-59071-2. DOI: 10.1007/3-540-59071-4_45 (siehe S. 120, 131, 287).

[Sch97] A. Schürr. „Programmed Graph Replacement Systems“. In: *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations*. World Scientific Publishing Co., Inc., 1997, S. 479–546 (siehe S. 47, 51, 52).

[Sei03] E. Seidewitz. „What models mean“. In: *Software, IEEE* 20.5 (Sep. 2003), S. 26–32. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231147 (siehe S. 16, 20).

[Sel+13] G. M. K. Selim, F. Büttner, J. R. Cordy, J. Dingel und S. Wang. „Automated Verification of Model Transformations in the Automotive Industry“. In: *Model-Driven Engineering Languages and Systems*. Hrsg. von A. Moreira, B. Schätz, J. Gray, A. Vallecillo und P. Clarke. Bd. 8107. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 690–706. ISBN: 978-3-642-41532-6. DOI: 10.1007/978-3-642-41533-3_42 (siehe S. 118).

[SG12] E. Syriani und J. Gray. „Challenges for Addressing Quality Factors in Model Transformation“. In: *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*. 2012, S. 929–937. DOI: 10.1109/ICST.2012.198 (siehe S. 116).

[SK03] S. Sendall und W. Kozaczynski. „Model Transformation: The Heart and Soul of Model-Driven Software Development“. In: *Software, IEEE* 20.5 (2003), S. 42–45. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231150 (siehe S. 3, 30, 35, 36).

[SL04] J. Steel und M. Lawley. „Model-based test driven development of the Tefkat model-transformation engine“. In: *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*. 2004, S. 151–160. DOI: 10.1109/ISSRE.2004.23 (siehe S. 125).

[SL05] A. Spillner und T. Linz. *Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester, Foundation Level nach ISTQB-Standard (3. Aufl.)* dpunkt.verlag, 2005. ISBN: 978-3-89864-358-0 (siehe S. 6).

[SMF99] M. Scheetz, A. von Mayrhauser und R. France. „Generating test cases from an OO model with an AI planning system“. In: *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*. 1999, S. 250–259. DOI: 10.1109/ISSRE.1999.809330 (siehe S. 107).

[SMS09] A. Simão, J. C. Maldonado und R. da Silva Bigonha. „A transformational language for mutant description“. In: *Computer Languages, Systems & Structures* 35.3 (2009), S. 322–339. ISSN: 1477-8424. DOI: 10.1016/j.cl.2008.10.001 (siehe S. 201).

[Soe+10] M. Soeken, R. Wille, M. Kuhlmann, M. Gogolla und R. Drechsler. „Verifying UML/OCL models using Boolean satisfiability“. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*. 2010, S. 1341–1344. DOI: 10.1109/DATE.2010.5457017 (siehe S. 112).

[Som95] I. Sommerville. *Software Engineering (5th Ed.)* Redwood City, Kalifornien, USA: Addison Wesley Publishing Co., Inc., 1995. ISBN: 0-201-42765-6 (siehe S. 4).

[Spe01] Specialist Interest Group in Software Testing (BCS SIGIST). *Standard for Software Component Testing, Working Draft 3.4*. British Computer Society, Apr. 2001 (siehe S. 84).

[SPK12] P. Skruch, M. Panek und B. Kowalczyk. „Model-Based Testing in Embedded Automotive Systems“. In: *Model-Based Testing for Embedded Systems*. Hrsg. von J. Zander, I. Schieferdecker und P. J. Mosterman. Computational Analysis, Synthesis, and Design of Dynamic Systems Series. CRC Press, 2012. Kap. 19, S. 545–575 (siehe S. 106).

[Ste+09] D. Steinberg, F. Budinsky, M. Paternostro und E. Merks. *EMF: Eclipse Modeling Framework, Second Edition*. 2. Auflage. Addison-Wesley Professional, 2009. ISBN: 978-0-321-33188-5 (siehe S. 22, 23, 323).

[Str08] M. Strecker. „Modeling and Verifying Graph Transformations in Proof Assistants“. In: *Electronic Notes in Theoretical Computer Science* 203.1 (2008). Proceedings of the Fourth International Workshop on Computing with Terms and Graphs (TERMGRAPH 2007), S. 135–148. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.03.039 (siehe S. 122).

[Str98] S. Strahringer. „Ein sprachbasierter Metamodellbegriff und seine Verallgemeinerung durch das Konzept des Metaisierungsprinzips“. In: Hrsg. von K. Pohl, A. Schürr und G. Vossen. Bd. Vol. 9. Modellierung ’98: Proceedings des GI-Workshops in Münster. März 1998 (siehe S. 20).

[Stu+07] I. Stürmer, I. Kreuz, W. Schäfer und A. Schürr. *Enhanced Simulink/Stateflow Model Transformation: The MATE Approach*. Proc. of MathWorks Automotive Conference (MAC 2007), Dearborn (MI), USA. Juni 2007 (siehe S. 75).

[Stü+07a] I. Stürmer, M. Conrad, H. Doerr und P. Pepper. „Systematic Testing of Model-Based Code Generators“. In: *Software Engineering, IEEE Transactions on* 33.9 (2007), S. 622–634. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70708 (siehe S. 6, 126, 133, 136, 138).

[Stü+07b] I. Stürmer, H. Dörr, H. Giese, U. Kelter, A. Schürr und A. Zündorf. *Das MATE Projekt – visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen*. Dagstuhl Seminar Modellbasierte Entwicklung eingebetteter Systeme. Jan. 2007 (siehe S. 35).

[SV03] Á. Schmidt und D. Varró. „CheckVML: A Tool for Model Checking Visual Modeling Languages“. In: *«UML» 2003 – The Unified Modeling Language. Modeling Languages and Applications*. Hrsg. von P. Stevens, J. Whittle und G. Booch. Bd. 2863. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, S. 92–95. ISBN: 978-3-540-20243-1. DOI: 10.1007/978-3-540-45221-8_8 (siehe S. 122).

[SV05] T. Stahl und M. Völter. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt-Verl. (Heidelberg), 2005 (siehe S. 15, 16, 29).

[SW08] D. A. Sadilek und S. Weißleder. „Testing Metamodels“. In: *Model Driven Architecture – Foundations and Applications*. Hrsg. von I. Schieferdecker und A. Hartman. Bd. 5095. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, S. 294–309. ISBN: 978-3-540-69095-5. DOI: 10.1007/978-3-540-69100-6_20 (siehe S. 109, 202).

[SZ09] D. Schuler und A. Zeller. „Javalanche: Efficient Mutation Testing for Java“. In: *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC/FSE '09. Amsterdam, Niederlande: ACM, 2009, S. 297–298. ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595750 (siehe S. 198).

[Tae+05] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, T. Levendovszky, U. Prange, D. Varró und S. Varró-Gyapay. „Model Transformations by Graph Transformations: A Comparative Study“. In: *Proceedings of the Model Transformations in Practice Workshop (co-located with MoDELS 2005)*. Montego Bay, Jamaika, 2005 (siehe S. 45).

[Tae+14] G. Taentzer, C. Ermel, P. Langer und M. Wimmer. „A fundamental approach to model versioning based on graph modifications: from theory to implementation“. In: *Software & Systems Modeling* 13.1 (2014), S. 239–272. ISSN: 1619-1366. DOI: 10.1007/s10270-012-0248-x (siehe S. 132, 203).

[Tae00] G. Taentzer. „AGG: A Tool Environment for Algebraic Graph Transformation“. In: *Applications of Graph Transformations with Industrial Relevance*. Hrsg. von M. Nagl, A. Schürr und M. Münch. Bd. 1779. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, S. 481–488. ISBN: 978-3-540-67658-4. DOI: 10.1007/3-540-45104-8_41 (siehe S. 46).

[Tae12] G. Taentzer. „Instance Generation from Type Graphs with Arbitrary Multiplicities“. In: *ECEASST* 47 (2012). Hrsg. von A. Fish und L. Lambers. http://journal.ub.tu-berlin.de/eceasst/article/view/727. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 43, 130).

[Tan09] A. S. Tanenbaum. *Moderne Betriebssysteme*. 3. aktualisierte Auflage. Pearson Studium - IT. Pearson Deutschland GmbH, Apr. 2009. ISBN: 978-3-8273-7342-7 (siehe S. 15).

[TC10] G. Tamura und A. Cleve. *A Comparison of Taxonomies for Model Transformation Languages*. Techn. Ber. `http://hal.inria.fr/inria-00488765` (zuletzt abgerufen am 3.4.2015), ID „inria-00488765“. März 2010 (siehe S. 31).

[Tis+09] M. Tisi, F. Jouault, P. Fraternali, S. Ceri und J. Bézivin. „On the Use of Higher-Order Model Transformations“. In: *Model Driven Architecture - Foundations and Applications*. Hrsg. von R. Paige, A. Hartman und A. Rensink. Bd. 5562. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 18–33. ISBN: 978-3-642-02673-7. DOI: `10.1007/978-3-642-02674-4_3` (siehe S. 238).

[TJ07] E. Torlak und D. Jackson. „Kodkod: A Relational Model Finder“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von O. Grumberg und M. Huth. Bd. 4424. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, S. 632–647. ISBN: 978-3-540-71208-4. DOI: `10.1007/978-3-540-71209-1_49` (siehe S. 113).

[Tre+07] C. Treude, S. Berlik, S. Wenzel und U. Kelter. „Difference Computation of Large Models“. In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. ESEC-FSE '07. Dubrovnik, Kroatien: ACM, 2007, S. 295–304. ISBN: 978-1-59593-811-4. DOI: `10.1145/1287624.1287665` (siehe S. 132).

[Tre96] J. Tretmans. „Test generation with inputs, outputs, and quiescence“. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Hrsg. von T. Margaria und B. Steffen. Bd. 1055. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, S. 127–146. ISBN: 978-3-540-61042-7. DOI: `10.1007/3-540-61042-1_42` (siehe S. 6).

[Tro+05] T. D. Trong, S. Ghosh, R. B. France, M. Hamilton und B. Wilkins. „UML-AnT: An Eclipse Plugin for Animating and Testing UML Designs“. In: *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*. eclipse '05. San Diego, Kalifornien: ACM, 2005, S. 120–124. ISBN: 1-59593-342-5. DOI: `10.1145/1117696.1117721` (siehe S. 109).

[TV10] J. Troya und A. Vallecillo. „Towards a Rewriting Logic Semantics for ATL“. In: *Theory and Practice of Model Transformations*. Hrsg. von L. Tratt und M. Gogolla. Bd. 6142. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, S. 230–244. ISBN: 978-3-642-13687-0. DOI: `10.1007/978-3-642-13688-7_16` (siehe S. 119).

[TV11] J. Troya und A. Vallecillo. „A Rewriting Logic Semantics for ATL“. In: *Journal of Object Technology* 10 (2011), 5:1–29. ISSN: 1660-1769. DOI: `10.5381/jot.2011.10.1.a5` (siehe S. 119).

[UK97] A. Ulrich und H. König. „Specification-based testing of concurrent systems“. In: *Formal Description Techniques and Protocol Specification, Testing and Verification*. Hrsg. von T. Mizuno, N. Shiratori, T. Higashino und A. Togashi. IFIP – The International Federation for Information Processing. Springer US, 1997, S. 7–22. ISBN: 978-1-4757-5260-1. DOI: 10.1007/978-0-387-35271-8_1 (siehe S. 106).

[UL07] M. Utting und B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., 2007. ISBN: 0-12-372501-1, 978-0-12-372501-1 (siehe S. 6, 102, 103, 106).

[Unh05] B. Unhelkar. *Verification and Validation for Quality of UML 2.0 Models*. Wiley-Interscience, Aug. 2005, S. 1–312. ISBN: 978-0-471-72783-5 (siehe S. 108).

[UPL12] M. Utting, A. Pretschner und B. Legeard. „A taxonomy of model-based testing approaches“. In: *Software Testing, Verification and Reliability* 22.5 (2012), S. 297–312. ISSN: 1099-1689. DOI: 10.1002/stvr.456 (siehe S. 102, 106).

[Val+12] A. Vallecillo, M. Gogolla, L. Burgueño, M. Wimmer und L. Hamann. „Formal Specification and Testing of Model Transformations“. In: *Formal Methods for Model-Driven Engineering*. Hrsg. von M. Bernardo, V. Cortellessa und A. Pierantonio. Bd. 7320. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 399–437. ISBN: 978-3-642-30981-6. DOI: 10.1007/978-3-642-30982-3_11 (siehe S. 127).

[Var+06] D. Varró, S. Varró–Gyapay, H. Ehrig, U. Prange und G. Taentzer. „Termination Analysis of Model Transformations by Petri Nets“. In: *Graph Transformations*. Hrsg. von A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro und G. Rozenberg. Bd. 4178. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, S. 260–274. ISBN: 978-3-540-38870-8. DOI: 10.1007/11841883_19 (siehe S. 121).

[Var04] D. Varró. „Automated formal verification of visual modeling languages by model checking“. In: *Software & Systems Modeling* 3.2 (2004), S. 85–113. ISSN: 1619-1366. DOI: 10.1007/s10270-003-0050-x (siehe S. 120, 122).

[VB07] D. Varró und A. Balogh. „The model transformation language of the VIATRA2 framework“. In: *Science of Computer Programming* 68.3 (2007). Special Issue on Model Transformation, S. 214–234. ISSN: 0167-6423. DOI: 10.1016/j.scico.2007.05.004 (siehe S. 51).

[VB10] M. F. Van Amstel und M. G. J. van den Brand. „Quality assessment of ATL model transformations using metrics“. In: *Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL 2010), Malaga, Spain (June 2010)*. Juni 2010 (siehe S. 115).

[VFV06] G. Varró, K. Friedl und D. Varró. „Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans“. In: *Electronic Notes in Theoretical Computer Science* 152 (2006). Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), S. 191–205. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.10.025 (siehe S. 70).

[Vig09] A. Vignaga. *Metrics for Measuring ATL Model Transformations*. Techn. Ber. http://swp.dcc.uchile.cl/TR/2009/TR_DCC-20090430-006.pdf (zuletzt abgerufen am 24.9.2014). Department of Computer Science, Universidad de Chile, 2009 (siehe S. 115).

[Vis01] E. Visser. „A Survey of Rewriting Strategies in Program Transformation Systems“. In: *Electronic Notes in Theoretical Computer Science* 57 (2001), S. 109–143. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)00270-1 (siehe S. 31).

[VMV09] R. Van Der Straeten, T. Mens und S. Van Baelen. „Challenges in Model-Driven Software Engineering“. In: *Models in Software Engineering*. Hrsg. von M. R. V. Chaudron. Bd. 5421. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, S. 35–47. ISBN: 978-3-642-01647-9. DOI: 10.1007/978-3-642-01648-6_4 (siehe S. 2, 4, 7).

[VP03] D. Varró und A. Pataricza. „Automated Formal Verification of Model Transformations“. In: *CSDUML 2003: Critical Systems Development in UML – Proceedings of the UML '03 Workshop*. Hrsg. von J. Jürjens, B. Rumpe, R. France und E. B. Fernandez. Techn. Ber. TUM-I0323. Technische Universität München, Sep. 2003, S. 63–78 (siehe S. 119, 132).

[VP04] D. Varró und A. Pataricza. „Generic and Meta-transformations for Model Transformation Engineering“. In: *«UML» 2004 — The Unified Modeling Language. Modeling Languages and Applications*. Hrsg. von T. Baar, A. Strohmeier, A. Moreira und S. Mellor. Bd. 3273. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, S. 290–304. ISBN: 978-3-540-23307-7. DOI: 10.1007/978-3-540-30187-5_21 (siehe S. 238).

[VSV05] G. Varró, A. Schürr und D. Varró. „Benchmarking for graph transformation“. In: *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*. 2005, S. 79–88. DOI: 10.1109/VLHCC.2005.23 (siehe S. 45).

[Wan+04] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li und G. Zheng. „Generating Test Cases from UML Activity Diagram based on Gray-box Method“. In: *Software Engineering Conference, 2004. 11th Asia-Pacific*. Nov. 2004, S. 284–291. DOI: 10.1109/APSEC.2004.55 (siehe S. 110).

[WAS14] M. Wieber, A. Anjorin und A. Schürr. „On the Usage of TGGs for Automated Model Transformation Testing“. In: *Theory and Practice of Model Transformations*. Hrsg. von D. Di Ruscio und D. Varró. Bd. 8568. Lecture Notes in Computer Science. Springer International Publishing, 2014, S. 1–16. ISBN: 978-3-319-08788-7. DOI: 10.1007/978-3-319-08789-4_1 (siehe S. 32, 129–131, 136, 250).

[WB13] M. Wimmer und L. Burgueño. „Testing M2T/T2M Transformations“. In: *Model-Driven Engineering Languages and Systems*. Hrsg. von A. Moreira, B. Schätz, J. Gray, A. Vallecillo und P. Clarke. Bd. 8107. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 203–219. ISBN: 978-3-642-41532-6. DOI: 10.1007/978-3-642-41533-3_13 (siehe S. 37, 133, 134).

[WC80] L. J. White und E. I. Cohen. „A Domain Strategy for Computer Program Testing“. In: *Software Engineering, IEEE Transactions on* SE-6.3 (Mai 1980), S. 247–257. ISSN: 0098-5589. DOI: 10.1109/TSE.1980.234486 (siehe S. 97).

[Wei10] S. Weißleder. „Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines“. http://d-nb.info/1011308983. Diss. Humboldt-Universität zu Berlin, 2010 (siehe S. 106).

[Wes96] R. S. Westfall. *Isaac Newton : eine Biographie*. (Aus dem Englischen übersetzt von Heiner Must). Spektrum, Akademischer Verlag, 1996. ISBN: 3-8274-0040-6 (siehe S. 105).

[Wim+09] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schönböck und W. Schwinger. „Right or Wrong? – Verification of Model Transformations using Colored Petri Nets“. In: *Proceedings of 9th OOPSLA Workshop on Domain-Specific Modeling*. 2009 (siehe S. 119, 129, 146, 205).

[Win+08] J. Winkelmann, G. Taentzer, K. Ehrig und J. M. Küster. „Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars“. In: *Electronic Notes in Theoretical Computer Science* 211 (2008). Proceedings of the Fifth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006), S. 159–170. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2008.04.038 (siehe S. 43, 130).

[WKC06] J. Wang, S.-K. Kim und D. Carrington. „Verifying metamodel coverage of model transformations“. In: *Software Engineering Conference, 2006. Australian*. 2006. DOI: 10.1109/ASWEC.2006.55 (siehe S. 126).

[WKC08] J. Wang, S.-K. Kim und D. Carrington. „Automatic Generation of Test Models for Model Transformations“. In: *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. März 2008, S. 432–440. DOI: 10.1109/ASWEC.2008.4483232 (siehe S. 130).

[WM97] R. Wilhelm und D. Maurer. *Übersetzerbau - Theorie, Konstruktion, Generierung*. 2. Aufl. ISBN: 978-3-540-61692-4. Springer, 1997. DOI: 10.1007/978-3-642-59081-8 (siehe S. 29, 30, 36, 39).

[WN69] D. E. Walker und L. M. Norton, Hrsg. *Proceedings of the 1st International Joint Conference on Artificial Intelligence, Washington, DC, May 1969*. William Kaufmann, 1969. ISBN: 0-934613-21-4 (siehe S. 372).

[WP01] A. W. Williams und R. L. Probert. „A measure for component interaction test coverage“. In: *Computer Systems and Applications, ACS/IEEE International Conference on. 2001*. 2001, S. 304–311. DOI: 10.1109/AICCSA.2001.934001 (siehe S. 130, 196).

[WS12] M. Wieber und A. Schürr. „Gray Box Coverage Criteria for Testing Graph Pattern Matching“. In: *ECEASST* 54 (2012). Hrsg. von C. Krause und B. Westfechtel. http://journal.ub.tu-berlin.de/eceasst/article/view/772. ISSN: 1863-2122. European Assoc. of Software Science and Technology (siehe S. 125).

[WS13] M. Wieber und A. Schürr. „Systematic Testing of Graph Transformations: A Practical Approach Based on Graph Patterns“. In: *Theory and Practice of Model Transformations*. Hrsg. von K. Duddy und G. Kappel. Bd. 7909. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 205–220. ISBN: 978-3-642-38882-8. DOI: 10.1007/978-3-642-38883-5_18 (siehe S. 145, 186, 194).

[WT94] J. Whittaker und G. Thomason Michael. „A Markov chain model for statistical software testing“. In: *Software Engineering, IEEE Transactions on* 20.10 (1994), S. 812–824. ISSN: 0098-5589. DOI: 10.1109/32.328991 (siehe S. 106).

[Wym12] O. Wyman. *FAST 2025: Future Automotive Industry Structure; eine Studie*. Materialien zur Automobilindustrie 45. Verband der Automobilindustrie (VDA), 2012 (siehe S. 1).

[XLL05] D. Xu, H. Li und C.-P. Lam. „Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams“. In: *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*. Dez. 2005. DOI: 10.1109/APSEC.2005.110 (siehe S. 111).

[XS05] Z. Xing und E. Stroulia. „UMLDiff: An Algorithm for Object-oriented Design Differencing“. In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ASE '05. Long Beach, Kalifornien, USA: ACM, 2005, S. 54–65. ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101919 (siehe S. 132).

[ZA06] I. A. Zualkernan und S. M. Abu-Naaj. „A web services-based architecture for mutation analysis of UML activity diagrams“. In: *GCC Conference (GCC), 2006 IEEE*. März 2006, S. 1–6. DOI: 10.1109/IEEEGCC.2006.5686219 (siehe S. 203).

[Zel+06] S. V. Zelenov, D. V. Silakov, A. K. Petrenko, M. Conrad und I. Fey. „Automatic Test Generation for Model-Based Code Generators“. In: *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*. Nov. 2006, S. 75–81. DOI: 10.1109/ISoLA.2006.70 (siehe S. 134).

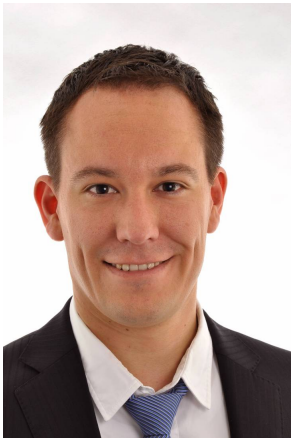
[ZSW99] A. Zündorf, A. Schürr und A. Winter. *Story Driven Modeling*. Techn. Ber. tr-ri-99-211. Universität Kassel, 1999 (siehe S. 53).

[Zun02] A. Zündorf. *Rigorous Object Oriented Software Development*. eine Entwurfsversion (v0.3) der Habil. ist online verfügbar: http://www.se.eecs.uni-kassel.de/fileadmin/se/publications/Zuen02.pdf (zuletzt abgerufen im März 2015). März 2002 (siehe S. 53, 54, 173).

[Zun96] A. Zündorf. „Graph pattern matching in PROGRES“. In: *Graph Grammars and Their Application to Computer Science*. Hrsg. von J. Cuny, H. Ehrig, G. Engels und G. Rozenberg. Bd. 1073. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, S. 454–468. ISBN: 978-3-540-61228-5. DOI: 10.1007/3-540-61228-9_105 (siehe S. 70).

[Zun98] A. Zündorf. *PROgrammierte GRaphErsetzungsSysteme*. DUV: Informatik. D82 (Diss. RWTH Aachen), Herausgegeben und eingeleitet von Manfred Nagl. Deutscher Universitäts-Verlag (Springer Fachm. Wiesbaden), 1998, S. I–XX, 1–374 (siehe S. 47, 52).

Lebenslauf



Persönliche Daten

Name	Martin Simon Wieber
Geboren am	3. Mai 1982
Geburtsort	Groß-Gerau, Hessen, Deutschland

Universität

06/2009 - 08/2014	Wissenschaftlicher Mitarbeiter am FG Echtzeitsysteme (Prof. Schürr), Institut für Datentechnik, FB18, Technische Universität Darmstadt
10/2004 - 03/2009	Hauptstudium an der Technischen Universität Darmstadt, Elektrotechnik und Informationstechnik (Diplomstudiengang) Abschluss mit Diplom zum Dipl.-Ing.
10/2002 - 09/2004	Grundstudium an der Technischen Universität Darmstadt, Elektrotechnik und Informationstechnik (Diplomstudiengang) Zwischenabschluss Vordiplom

Schule

08/1992 - 06/2001	Justus-Liebig-Schule, Darmstadt (Gymnasium)
08/1988 - 06/1992	Georg-August-Zinn-Schule, Darmstadt (Grundschule)

Publikationen

2014 M. Wieber, A. Anjorin und A. Schürr. „On the Usage of TGGs for Automated Model Transformation Testing“. In: *Theory and Practice of Model Transformations*. Hrsg. von D. Di Ruscio und D. Varró. Bd. 8568. Lecture Notes in Computer Science. Springer International Publishing, 2014, S. 1–16. ISBN: 978-3-319-08788-7. DOI: 10.1007/978-3-319-08789-4_1.

2013 G. Varró, F. Deckwerth, M. Wieber und A. Schürr. „An algorithm for generating model-sensitive search plans for pattern matching on EMF models“. In: *Software & Systems Modeling* (2013), S. 1–25. ISSN: 1619-1366. DOI: 10.1007/s10270-013-0372-2.

2013 M. Wieber und A. Schürr. „Systematic Testing of Graph Transformations: A Practical Approach Based on Graph Patterns“. In: *Theory and Practice of Model Transformations*. Hrsg. von K. Duddy und G. Kappel. Bd. 7909. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, S. 205–220. ISBN: 978-3-642-38882-8. DOI: 10.1007/978-3-642-38883-5_18.

2012 G. Varró, F. Deckwerth, M. Wieber und A. Schürr. „An Algorithm for Generating Model-Sensitive Search Plans for EMF Models“. In: *Theory and Practice of Model Transformations*. Hrsg. von Z. Hu und J. Lara. Bd. 7307. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, S. 224–239. ISBN: 978-3-642-30475-0. DOI: 10.1007/978-3-642-30476-7_15.

2012 M. Wieber und A. Schürr. „Gray Box Coverage Criteria for Testing Graph Pattern Matching“. In: *Proc. of Graph-Based Tools 2012*. Hrsg. von C. Krause und B. Westfechtel. Bd. 54. ECEASST. http://journal.ub.tu-berlin.de/eceasst/article/view/772. 2012. European Assoc. of Software Science and Technology.

13. Juli 2015, Darmstadt